DOS-Referenz

VERSION 2.5

Borland® C++

Borland C++ 4.5

DOS-Referenz

BORLAND

Copyright © 1987, 1993 by Borland International, Inc.
Alle Rechte vorbehalten.

Alle Borland-Produkte sind Warenzeichen oder eingetragene Warenzeichen von Borland International, Inc. Andere aufgeführte Waren- und Produktbezeichnungen sind Warenzeichen oder eingetragene Warenzeichen der jeweiligen Rechteinhaber.

Einschränkung der Gewährleistung:
Es wird keine Garantie für die Richtigkeit des Inhaltes dieses Handbuches übernommen. Da sich Fehler, trotz aller Bemühungen, nie vollständig vermeiden lassen, sind wir für Hinweise jederzeit dankbar.

Herausgeber:

Borland GmbH
Monzastraße 4c
63225 Langen

Printed in Ireland.

Inhalt

Einführung 9
Was Sie in diesem Buch finden 9

Kapitel 1
Speicherverwaltung unter DOS 11
Speicherplatzprobleme 11
Speichermodelle 11
Die Register des 8086 12
Allzweckregister 12
Segmentregister 13
Spezialregister 13
Das Flag-Register 13
Segmentierung des Speichers 14
Adreßberechnung 15
Zeigertypen 16
Near-Zeiger 16
Far-Zeiger 16
Huge-Zeiger 17
Die sechs Speichermodelle 18
Das Programmieren mit gemischten Speichermodellen: Modifizierer 23
Segment-Zeiger 24
Deklarieren von far-Objekten 25
Explizite Funktionsdeklarationen mit near und far 25
Explizite Zeigerdeklarationen mit near, far und huge 26
Direktes Setzen von Zeigern (Segment:Offset-Adresse) 27
Bibliotheksdateien 28
Linken von Modulen mit unterschiedlichen Speichermodellen 28
Overlays (VROOM) für DOS 29
Die Arbeitsweise von Overlays 30
Voraussetzungen für die Erzeugung von Overlays 32
Overlays und Exception-Behandlung . . . 33
Verwenden von Overlays 34
Gestalten von Programmen mit Overlays 35
Debuggen von Overlays 36
Externe Assembler-Routinen in Overlays 36
Aus- und Einlagerung 37

Kapitel 2
Arithmetik 39
Ein-/Ausgabe von Gleitkommawerten 39
Gleitkomma-Optionen 39
Emulation des 80x87 40
Verwendung des 80x87-Codes 40
Kein Gleitkomma-Code 40
Schnelle Gleitkomma-Operationen 41
Die Umgebungsvariable 87 41
Register und Koprozessor 42
Fehlerbedingungen bei Gleitkomma-Operationen 42
Komplexe Arithmetik 43
BCD-Arithmetik 44
Konvertierung von BCD-Zahlen 45
Anzahl der Dezimalstellen 46

Kapitel 3
Bildschirmgrafik 47
Erläuterungen zu den Video-Modi 47
Text- und Grafikfenster 48
Programmieren im Grafikmodus 49
Die Funktionen der Grafikbibliothek . . . 49

Kapitel 4
Grafikfunktionen 65
arc 65
bar 66
bar3d 66
circle 67
cleardevice 67
clearviewport 67

closegraph 68
detectgraph 68
drawpoly 70
ellipse 70
fillellipse 71
fillpoly 71
floodfill 71
getarccoords 72
getaspectratio 72
getbkcolor 73
getcolor 73
getdefaultpalette 73
getdrivername 74
getfillpattern 74
getfillsettings 74
getgraphmode 76
getimage 76
getlinesettings 76
getmaxcolor 77
getmaxmode 78
getmaxx 78
getmaxy 79
getmodename 79
getmoderange 79
getpalette 80
getpalettesize 80
getpixel 81
gettextsettings 81
getviewsettings 82
getx 82
gety 82
graphdefaults 83
grapherrormsg 83
_graphfreemem 84
_graphgetmem 84
graphresult 84
imagesize 86
initgraph 86
installuserdriver 89
installuserfont 90
line 91
linerel 91
lineto 91
moverel 92
moveto 92
outtext 92
outtextxy 93
pieslice 93
putimage 94
putpixel 95
rectangle 95
registerbgidriver 95
registerbgifont 96
restorecrtmode 96
sector 97
setactivepage 97
setallpalette 98
setaspectratio 99
setbkcolor 100
setcolor 101
setfillpattern 101
setfillstyle 102
setgraphbufsize 103
setgraphmode 103
setlinestyle 104
setpalette 105
setrgbpalette 106
settextjustify 107
settextstyle 107
setusercharsize 109
setviewport 109
setvisualpage 110
setwritemode 110
textheight 111
textwidth 111

Kapitel 5
DOS-Funktionen 113

absread 113
abswrite 114
allocmem, _dos_allocmem 114
bioscom 115
biosdisk 117
_bios_disk 120
bioskey 122
_bios_keybrd 123
biosprint 124
_bios_printer 125
_bios_serialcom 126

brk 128
coreleft 128
delay 129
farcoreleft 129
farheapcheck 129
farheapcheckfree 130
farheapchecknode 130
farheapfillfree 131
farheapwalk 131
freemem, _dos_freemem 132
harderr, hardresume, hardretn 132
_harderr 133
_hardresume 134
_hardretn 135
keep, _dos_keep 135
nosound 136
_OvrInitEms 136
_OvrInitExt 137
randbrd 137
randbwr 138
sbrk 139
setblock, _dos_setblock 139
sound 140

Anhang A
DOS-Bibliotheken 141

Die Laufzeitbibliotheken 141
Die DOS-Bibliotheken 142
Grafikroutinen 142
Schnittstellenroutinen 143
Routinen zur Speicherverwaltung 144
Verschiedene Routinen 144

Anhang B
Globale Variablen für
DOS-Anwendungen 145

_heaplen 145
_ovrbuffer 146
_stklen 147

Index 149

Bilder

1.1: Die Register des 8086 18
1.3: Segmentierung im Modell *Tiny* 25
1.4: Segmentierung im Modell Small 26
1.5: Segmentierung im Modell Medium 26
1.6: Segmentierung im Modell Compact 27
1.7: Segmentierung im Modell Large 27
1.8: Segmentierung im Modell Huge 28
1.9: Die Belegung des Speichers durch Overlays 28

Tabellen

1.1: Speichermodelle 28
1.2: Standard-Zeigertypen versus Speichermodelle 30
3.1: Statusabfragefunktionen 67
4.1: Konstanten für *detectgraph* 74
4.2: Grafiktreiber 75
4.3: Grafiktreiber-Konstanten 93
4.4: Farbpaletten 94
4.5: Grafikmodi 94
4.6: Farb 105
A.1: Zusammenfassung der DOS-Laufzeitbibliotheken 148

Einführung

Einen vollständigen Überblick über die Borland C++ Dokumentation finden Sie im Einführungsteil des *Benutzerhandbuch*.

Dieses Handbuch enthält Informationen über die Entwicklung von 16-Bit Anwendungen, die unter DOS laufen sollen. In den folgenden Handbüchern finden Sie weitere Informationen, die ebenfalls für die Entwicklung von DOS-Anwendungen wichtig sind.

- Im *Benutzerhandbuch* finden Sie eine Beschreibung aller Programmieroptionen, die Sie bei der Anwendungsentwicklung für die verschiedenen, von Borland C++ unterstützten Zielplattformen verwenden können.
- Im *Programmierhandbuch* werden die Borland C++ Implementation und die Erweiterungen der Sprachen C und C++ beschrieben. Viele der dort zu findenden Informationen (wie etwa über Exception-Behandlung, RTTI und weitere neue Erweiterungen der Sprache C++) sind auch in der 16-Bit DOS-Programmierung verwendbar.
- Das *Referenzhandbuch* enthält eine vollständige Referenz aller Borland C++ Routinen – einschließlich aller Klassen, Funktionen und Makros. Eine Markierung in den Portabilitätstabellen weist darauf hin, daß der entsprechende Eintrag auch für DOS-Programme zur Verfügung steht.

Die in diesem Handbuch verwendeten Schriftarten und Symbole sind im *Benutzerhandbuch* erläutert.

Was Sie in diesem Buch finden

Kapitel 1: Speicherverwaltung unter DOS beschreibt Speichermodelle, Overlays und Programmierung mit mehreren Speichermodellen. Beachten Sie, daß Sie bei der DOS-Programmierung alle sechs Speichermodelle verwenden können. Die Speichermodelle Tiny und Huge werden in Windows-Anwendungen jedoch nicht unterstützt. Overlays werden nur in DOS-Anwendungen unterstützt.

Kapitel 2: Arithmetik beinhaltet Gleitkommaberechnungen und die Verwendung der Klassen *bcd* und *complex*. Viele dieser Informationen beziehen sich nur auf DOS-Anwendungen. Die Klassen *bcd* und *complex* sind jedoch auch unter Windows und OS/2 verfügbar.

Kapitel 3: Bildschirmgrafik beschreibt die Behandlung von Grafik in Borland C++. Die in diesem Kapitel enthaltenen Informationen beziehen sich ausschließlich auf 16-Bit DOS-Anwendungen.

Kapitel 4: Grafikfunktionen enthält eine Referenz aller in der Header-Datei graphics.h deklarierten Funktionen. Die in diesem Kapitel vorgestellten Funktionen sind nur für 16-Bit DOS-Anwendungen verfügbar. Beispielprogramme zu den Funktionen finden Sie in der Online-Hilfe.

Kapitel 5: DOS-Funktionen beinhaltet alle Funktionen, die nur für 16-Bit DOS-Anwendungen zur Verfügung stehen. Es gibt zahlreiche weitere Funktionen und Klassen, die bei der DOS-Programmierung verwendet werden können (und zusätzlich auch auf anderen Plattformen verfügbar sind). Die Beschreibung dieser Funktionen finden Sie im *Referenzhandbuch*. In der Online-Hilfe finden Sie zahlreiche Beispielprogramme für die Funktionen, die in diesem Kapitel und im *Referenzhandbuch* beschrieben sind.

Anhang A: DOS-Bibliotheken enthält einen Überblick über die Bibliotheken, die ausschließlich für 16-Bit DOS-Anwendungen zur Verfügung stehen.

Anhang B: Globale Variablen für DOS-Anwendungen enthält einen Überblick über die globalen Variablen, die ausschließlich in 16-Bit DOS-Anwendungen zur Verfügung stehen.

K A P I T E L 1

Speicherverwaltung unter DOS

Das vorliegende Kapitel befaßt sich mit den folgenden drei Themenbereichen:

- Der Vorgehensweise bei *Speicherplatzproblemen* (Out-of-memory-Fehler).
- Was Speichermodelle sind, nach welchen Kriterien sie ausgewählt werden, und wie sie sinnvoll eingesetzt werden.
- Der Arbeitsweise und dem Einsatz von *Overlays*.
- Wie man mit *Overlays* umgeht bei Modulen mit Konstrukten zur Exception-Behandlung.

Speicherplatzprobleme

Borland C++ erzeugt keine Zwischendateien auf der Festplatte (es werden nur .OBJ-Dateien auf die Festplatte geschrieben), da die gesamte Compilierung im RAM stattfindet. Aus diesem Grund erhalten Sie die Meldung `Out of memory...`, wenn die Kapazität des Hauptspeichers während der Compilierung nicht ausreicht.

Dieses Problem kann auf zwei Arten gelöst werden: durch die Verkleinerung der Funktionen oder durch die Aufteilung einer Datei mit umfangreichen Funktionen.

Speichermodelle

Auf Seite 18 sehen Sie eine Übersicht über die Speichermodelle.

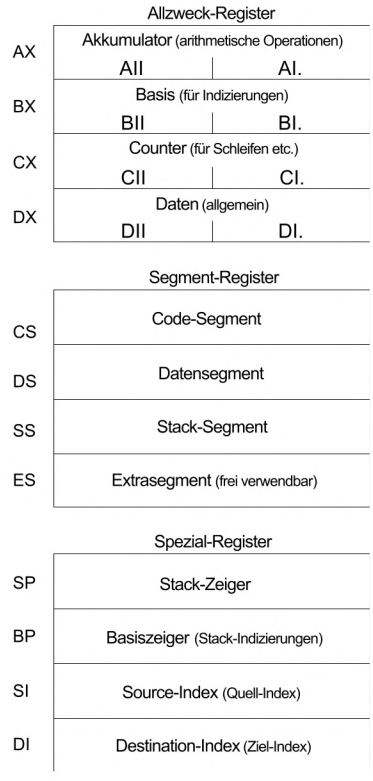
In Borland C++ gibt es zur Abstimmung auf den jeweiligen Programmumfang sechs verschiedene Speichermodelle, die den vorhandenen Hauptspeicher unterschiedlich nutzen. Der Grund für diese Unterteilung liegt im Aufbau des Computers. Dieser verwendet einen Zentralprozessor (CPU) der Intel iAPx86-Familie (80286, 80386, 80486 oder Pentium).

Bei den weiteren Ausführungen wird die Bezeichnung 8086 als Sammelbegriff verwendet.

Die Register des 8086

Die Abbildung 1.1 zeigt die Register des Prozessors 8086. Es gibt weitere Register, auf die Borland C++ aber nicht direkt zugreifen kann. Sie sind daher in der Abbildung nicht aufgeführt.

Abbildung 1.1:
Die Register des 8086



Allzweckregister

Allzweckregister werden vor allem zur Aufnahme und Manipulation von Daten verwendet. Jedes Register hat seine ganz speziellen Aufgaben. Z.B.:

- AX ist das zentrale Register für arithmetische Operationen;
- BX dient als Indexregister;
- CX wird von Schleifen und einigen String-Befehlen genutzt;
- DX wird bei einigen speziellen arithmetischen Operationen implizit benutzt.

Sehr viele Operationen können in allen Registern ausgeführt werden, das heißt, die Register sind meist frei wählbar.

Segmentregister

Die Segmentregister speichern die Startadressen der vier Segmente. Der 16-Bit-Wert in einem Segmentregister wird um 4 Bits nach links geschoben (Multiplikation mit 16), um die vollständige 20-Bit-Adresse des Segments darzustellen.

Spezialregister

Der 8086 besitzt mehrere Spezialregister:

- Die Register SI und DI verhalten sich im Prinzip wie Allzweckregister, werden aber auch als Indexregister eingesetzt. Borland C++ verwendet sie außerdem für Registervariablen.
- Das Register SP (Stackzeiger) zeigt auf die momentane Spitze des Stack und ist ein Offset innerhalb des Stacksegments.
- Das Register BP ist ein nachgeordneter Stackzeiger und wird für die Indizierung des Stacks benutzt, um Argumente oder automatische Variablen aufzufinden.

Borland C++ Funktionen verwenden das Register BP als Basisadresse für Argumente und automatische Variablen. Parameter haben, je nach Speichermodell, unterschiedliche positive Offsets von BP. Das Register zeigt immer auf den vorher in ihm gespeicherten Wert. Funktionen ohne Argumente machen von BP keinen Gebrauch (und sichern es auch nicht), falls die Option *Standard Stack Frame* ausgeschaltet ist.

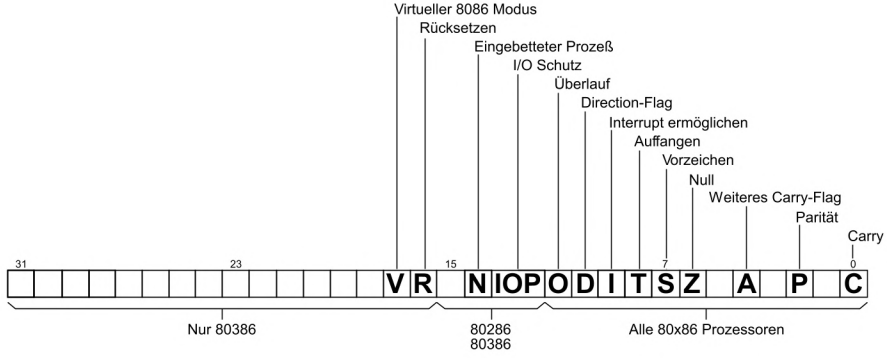
Automatische Variablen haben negative Offsets von BP, die davon abhängen, wieviel Speicher bereits lokalen Variablen zugewiesen wurde.

Das Flag-Register

Das 16-Bit Flag-Register enthält alle wichtigen Informationen über den Status des 8086 und die Ergebnisse bereits ausgeführter Befehle.

Um zum Beispiel festzustellen, ob eine Subtraktion das Ergebnis Null hatte, überprüfen Sie sofort nach dem Befehl das *Zero-Flag* (das Z-Bit im Flag-Register). Das Ergebnis ist Null, wenn das Flag gesetzt worden ist. Andere Flags (zum Beispiel das *Carry*- oder das *Overflow-Flag*) beschreiben ebenfalls Ergebnisse von arithmetischen oder logischen Operationen.

Abbildung 1.2:
Flag-Register des 8086



Weitere Flags steuern die Operationsmodi des 8086. So legt das *Direction-Flag* die Richtung von String-Befehlen fest, und das *Interrupt-Flag* bestimmt, ob externe Hardware (etwa die Tastatur oder ein Modem) den Programmablauf vorübergehend unterbrechen dürfen, damit dringende Aufgaben erledigt werden können. Das *Trap-Flag* kann nur von Software genutzt werden, die zum Debuggen anderer Software eingesetzt wird.

Das Flag-Register wird normalerweise nicht direkt modifiziert oder gelesen. Die Kontrolle der Flags erfolgt über spezielle Assembler-Befehle wie **CLD**, **STI** oder **CMC** und über arithmetische und logische Befehle, die bestimmte Flags modifizieren. Umgekehrt beeinflußt der Inhalt bestimmter Bits des Flag-Registers die Wirkungsweise von Befehlen wie **JZ**, **RCR**, oder **MOVSB**. Das Flag-Register dient demnach nicht als Speicher, sondern enthält die Status- und Steuerungsdaten für den 8086.

Segmentierung des Speichers

Der Intel-Mikroprozessor 8086 hat eine *segmentierte Speicherarchitektur*. Er kann zwar auf insgesamt 1 Megabyte Speicher zugreifen, kann aber immer nur 64 KByte dieses Speichers direkt adressieren.

Ein 64-KByte-Abschnitt des Speichers wird Segment genannt (daher der Begriff »segmentierte Speicherarchitektur«).

- Der 8086 verwaltet permanent vier verschiedene Segmente: ein *Code-*, ein *Daten-*, ein *Stack-* und ein *Extrasegment*. Im Code-Segment stehen die Maschinenbefehle, im Datensegment die Informationen, im Stack-Segment der Stack und das Extrasegment steht für weitere Daten zur Verfügung.
- Für diese vier Segmente steht jeweils ein eigenes 16-Bit-Register zur Verfügung, die ihre Adresse enthalten. Diese sind: CS, DS, SS und ES, die jeweils auf das *Code-*, *Daten-*, *Stack-* und *Extrasegment* zeigen.

- Ein Segment kann an (fast) jeder beliebigen Stelle im Speicher stehen.Aus Gründen, die später näher erläutert werden, muß ein Segment an einer Adresse beginnen, die ohne Rest durch 16 (dezimal) teilbar ist.

Adreßberechnung

Dieser Abschnitt bezieht sich nur auf den Real Mode unter DOS. Für Windows im Standard- und erweiterten 386er-Modus spielen diese Ausführungen keine Rolle.

Eine vollständige Adresse des 8086 setzt sich aus zwei 16-Bit-Werten zusammen: der Segmentadresse und dem Offset. Angenommen, die Segmentadresse – der Wert im DS-Register – ist 2F84 (hexadezimal), und es soll die tatsächliche Adresse für Daten berechnet werden, die vom Anfang des Datensegments einen Offset von 0532 (hexadezimal) haben: Wie sieht dann die Ausführung aus?

Die Adreßberechnung geht folgendermaßen vor sich: Der Wert des Segmentregisters wird um 4-Bit-Positionen nach links verschoben (entspricht einer Hex-Ziffer), dann wird der Offset hinzuaddiert.
Das Ergebnis ist eine vollständige 20-Bit-Adresse. Dazu ein Beispiel:

```
DS-Register (verschoben): 0010 1111 1000 0100 0000 = 2F840
Offset:                   0000 0101 0011 0010 = 00532
-----------------------------------------------------------
Adresse:                  0010 1111 1101 0111 0010 = 2FD72
```

Ein Speicherbereich von 16 Bytes wird auch als *Paragraph* bezeichnet. Der Beginn eines Segments liegt demnach immer an einer Paragraphengrenze.

Die Startadresse eines Segments ist immer eine 20-Bit-Zahl. Da ein Segmentregister aber nur 16-Bit speichern kann, werden die untersten 4 Bits immer zu 0 angenommen. Deshalb können Segmente nur an Adressen beginnen, die ein ganzzahliges Vielfaches von 16 sind, an einer Adresse, deren letzte vier Bits (die letzte Hex-Ziffer) Null sind. Enthält z.B. das DS-Register den Wert 2F84, beginnt das Datensegment bei der Adresse 2F840.

Die Standardnotation für eine Adresse hat die Form *Segment:Offset*. Für die obige Adresse können Sie demnach *2F84:0532* schreiben. Eine *Segment:Offset*-Kombination muß nicht eindeutig sein, da sich Offsets überlappen können. So läßt sich ein und dieselbe absolute Adresse durch eine Vielzahl von Kombinationen darstellen:

```
0000:0123
0002:0103
0008:00A3
0010:0023
0012:0003
```

Da sich Segmente überlappen können (aber nicht müssen), ist es möglich, alle vier Segmente (Code-, Daten-, Stack- und Extrasegment) an derselben Adresse beginnen zu lassen. Für Programmcode, Daten und Stack stünden dann allerdings nur 64 KByte zur Verfügung.

Zeigertypen

Normalerweise legt Borland C++ den Typ der Zeiger für Code und Daten über das jeweilige Speichermodell fest. Durch entsprechende Modifizierer bei der Deklaration lassen sich Zeigertypen aber auch unabhängig vom Modell festlegen: Es gibt folgende Zeigertypen: **near** (16 Bit), **far** (32 Bit), **huge** (32 Bit) und **segment** (16 Bit).

Near-Zeiger

Ein **near**-Zeiger (16 Bit) ist bei der Berechnung seiner Adresse von einem Segmentregister abhängig. Zum Beispiel bekommen Zeiger auf Funktionen den aktuellen nach links verschobenen Inhalt des Code-Segments (CS) vorangestellt, Zeiger auf Daten des Datensegment-Registers (DS). Als **near** deklarierte Zeiger sind einfach zu manipulieren: Arithmetische Operationen (z.B. eine Addition) finden ohne Berücksichtigung der Segmentadresse statt.

Far-Zeiger

Ein **far**-Zeiger (32 Bit) enthält nicht nur den Offset des Segments, sondern auch die Segmentadresse (als weiteren 16-Bit-Wert). Dieser wird nach links verschoben und zum Offset addiert. Auf Funktionen angewandt, erlauben **far**-Zeiger mehrere Code-Segmente (also Programme mit mehr als 64 KByte). Mit **far**-Zeigern können auch Daten mit mehr als 64 KByte adressiert werden.

Die Verwendung von **far**-Zeigern für Daten kann zu Problemen bei der Zeigermanipulation führen. Wie im Abschnitt »Adreßberechnung« erläutert, kann dieselbe absolute Adresse durch mehrere Kombinationen aus *Segment:Offset* bezeichnet werden – z.B. bezeichnen die drei **far**-Zeiger mit den Werten 0000:0120, 0010:0020 und 0012:0000 alle dieselbe 20-Bit-Adresse. Wenn wir davon ausgehen, daß drei Zeigervariablen mit den Namen *a*, *b* und *c* deklariert sind, die diese Werte enthalten, dann haben sämtliche der folgenden Vergleiche das Ergebnis *0* (*false*).

```
if (a == b) ...
if (b == c) ...
if (a == c) ...
```

Ähnliche Probleme ergeben sich bei Vergleichen von **far**-Zeigern mit den Operatoren **>**, **>=**, **<=**, **<**. Hier vergleicht Borland C++ nur die Offset-Anteile (als **unsigned**) miteinander und liefert in allen drei folgenden Vergleichen *1* (*true*) zurück, wenn *a*, *b* und *c* die oben gezeigten Werte haben:

```
if (a > b) ...
if (b > c) ...
if (a > c) ...
```

Die Operatoren gleich (**==**) und ungleich (**!=**) werten den 32-Bit-Wert als **unsigned long** (nicht als vollständige Speicheradresse). Die Vergleichsoperatoren (**<=**, **>=**, **<** und **>**) verwenden nur den Offset-Anteil.

Die Operatoren **==** und **!=** benötigen alle 32 Bit, damit ein Vergleich mit dem NULL-Zeiger (0000:0000) stattfinden kann. Würde nur der Offset-Anteil für den Vergleich benutzt, würde jeder Zeigerwert der Form *Segment:0000* als mit dem NULL-Zeiger identisch behandelt.

Beim Addieren von Werten zu einem **far**-Zeiger wird nur der Offset-Anteil des Zeigers geändert. Bei Erhöhungen über FFFF (den Maximalwert) hinaus beginnt die Zählung einfach wieder am Beginn des Segments. Zum Beispiel hat die Erhöhung des Zeigerwerts 5031:FFFF um 1 das Ergebnis 5031:0000 zur Folge (nicht 6031:0000). Analog dazu erzeugt die Subtraktion des Zeigerwertes 5031:0000 um 1 das Ergebnis 5031:FFFF (nicht 5030:000F).

Programme, die mit Zeigervergleichen arbeiten, sollten deshalb entweder **near**-Zeiger (diese arbeiten nur innerhalb eines Segments) oder **huge**-Zeiger verwenden; diese werden als nächstes beschrieben.

Huge-Zeiger

huge-Zeiger werden ebenfalls mit 32 Bit dargestellt, enthalten also wie **far**-Zeiger einen Wert der Form *Segment:Offset*. Im Unterschied zu **far**-Zeigern liegen sie immer in *normalisierter* Form vor. Damit sind die beschriebenen Probleme gelöst.

Ein *normalisierter* Zeiger ist ein 32-Bit-Zeiger, der in seiner Segmentadresse soviel von seinem Wert enthält, wie möglich. Da ein Segment nach jeweils 16 Bytes (10h) beginnen kann, hat der Offset einen Wert im Bereich von 0 bis 15 (0 bis F im Hexadezimalsystem).

Zur Normalisierung eines Zeigers wird dieser in seine 20-Bit-Adresse konvertiert. Dann werden die 4 rechten Bits für den Offset und die 16 linken Bits für die Segmentadresse verwendet. Aus einem Wert wie 2F84:0532 wird also zuerst die absolute Adresse 2FD72 berechnet; die nachfolgende Unterteilung dieser Adresse in Segment und Offset ergibt 2FD7:0002. Einige weitere Vergleiche von Zeigerwerten und ihren normalisierten Entsprechungen:

```
0000:0123     0012:0003
0040:0056     0045:0006
500D:9407     594D:0007
7418:D03F     811B:000F
```

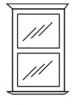
Aus drei Gründen müssen **huge**-Zeiger immer normalisiert sein:

1. Für jede Speicheradresse gibt es nur eine einzige Kombination *Segment:Offset*. Vergleiche mit den Operatoren **==** und **!=** liefern deshalb immer korrekte Ergebnisse.

2. Die Operatoren **>**, **>=**, **<=** und **<** benutzen den gesamten 32-Bit-Wert von **huge**-Zeigern. Die Normalisierung stellt sicher, daß Vergleiche mit diesen Operatoren korrekte Ergebnisse liefern.
3. Aufgrund der Normalisierung beginnt die Zählung im Offset-Anteil eines **huge**-Zeigers nach jeweils 16 Werten von neuem. Allerdings wird – im Gegensatz zu **far**-Zeigern – der Segmentanteil angepaßt. Wird z.B. 811B:000F um eins erhöht, lautet das Ergebnis 811C:0000. Analog dazu liefert die Dekrementierung von 811C:0000 um 1 das Ergebnis 811B:000F. Aufgrund der Normalisierung ist über **huge**-Zeiger auch die Manipulation von Datenstrukturen möglich, deren Größe 64 KByte übersteigt. Dadurch ist es möglich, daß bei einem **huge**-Array von Strukturen, das größer als 64 KByte ist, die Indizierung in das Array und die Auswahl eines **struct**-Feldes auch bei Strukturen mit beliebiger Größe funktioniert.

Die Verwendung von **huge**-Zeigern hat aber auch einen Nachteil: zusätzlichen Overhead. Bei jeder Veränderung eines **huge**-Zeigers muß ein Unterprogramm zur Normalisierung des Ergebnisses aufgerufen werden. Programme, die sehr viel mit Zeigern dieses Formats arbeiten, werden deshalb bedeutend langsamer ausgeführt als solche mit **far**- und **near**-Zeigern.

Die sechs Speichermodelle



Borland C++ stellt sechs verschiedene Speichermodelle zur Verfügung: *Tiny*, *Small*, *Medium*, *Compact*, *Large* und *Huge*. Die Wahl des Modells hängt von den Erfordernissen Ihres Programms ab. (In Kapitel 8, »Erstellen einer Windows-Anwendung« erfahren Sie Näheres zur Auswahl von Speichermodellen für Windows-Anwendungen.) Die folgende Übersicht gibt eine Zusammenfassung der Eigenschaften aller sechs Modelle:

- *Tiny* ist das kleinste Speichermodell. Alle vier Segmentregister (CS, DS, SS und ES) werden auf dieselbe Adresse gesetzt – für Programmcode, Daten und Stack stehen also maximal 64 KByte zur Verfügung. Sowohl für Funktionsaufrufe als auch für Daten werden **near**-Zeiger verwendet. Mit diesem Modell erzeugte Programme sind die Lösung, wenn es ausschließlich um einen minimalen Verbrauch an Speicherplatz geht – außerdem lassen sich (wenn sie mit der Kommandozeilenoption **/t** gelinkt werden) Dateien des Typs .COM erzeugen.
- *Small* arbeitet mit getrennten Segmenten für Code und Daten: Für den Programmcode stehen (maximal) 64 KByte zur Verfügung, für Daten und Stack weitere 64 KByte. Für Funktionsaufrufe und für Daten werden stets **near**-Zeiger verwendet. Dieses Modell eignet sich am ehesten für mittlere Programme.
- *Medium* benutzt **far**-Zeiger nur für die Adressierung von Funktionen. Der Code kann damit theoretisch ein MByte umfassen, für Daten (ein-

schließlich Stack) stehen nur 64 KByte zur Verfügung. *Medium* eignet sich am ehesten für sehr große Programme, die wenige Daten verwalten.

- *Compact* ist das Gegenstück zu *Medium* und verwendet **far**-Zeiger für Daten, nicht aber für Code. Der Programmcode ist auf 64 KByte begrenzt, für Daten steht der Rest des Speichers (theoretisch bis zu einem Megabyte) zur Verfügung. Das Modell *Compact* eignet sich am ehesten für kleine Programme mit großen Datenmengen.
- Large verwendet **far**-Zeiger sowohl für Code als auch für Daten (jeweils 1 MByte). Die Speichermodelle *Large* und *Huge* sollten nur für sehr große Programme verwendet werden.
- *Huge* verwendet ebenfalls **far**-Zeiger für Code und Daten. Dieses Modell erlaubt als einziges die Deklaration von mehr als 64 KByte statischer Daten. Normalerweise begrenzt Borland C++ die Größe statischer Daten auf 64 KByte.

Zur Auswahl des Speichermodells können Sie entweder entsprechende Menüpunkte der integrierten Entwicklungsumgebung (IDE) auswählen oder die Kommandozeilenversion des Compilers von Borland C++ mit den entsprechenden Parametern aufrufen.

Die Abbildungen 1.3. bis 1.8. auf den folgenden Seiten veranschaulichen, in welcher Weise der Speicher des 8086 bei Verwendung der verschiedenen Modelle aufgeteilt wird.

Abbildung 1.3: Segmentierung im Modell *Tiny*

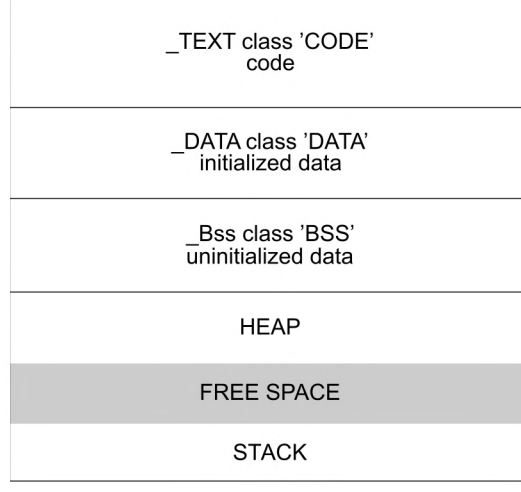


Abbildung 1.4:
Segmentierung im Modell Small

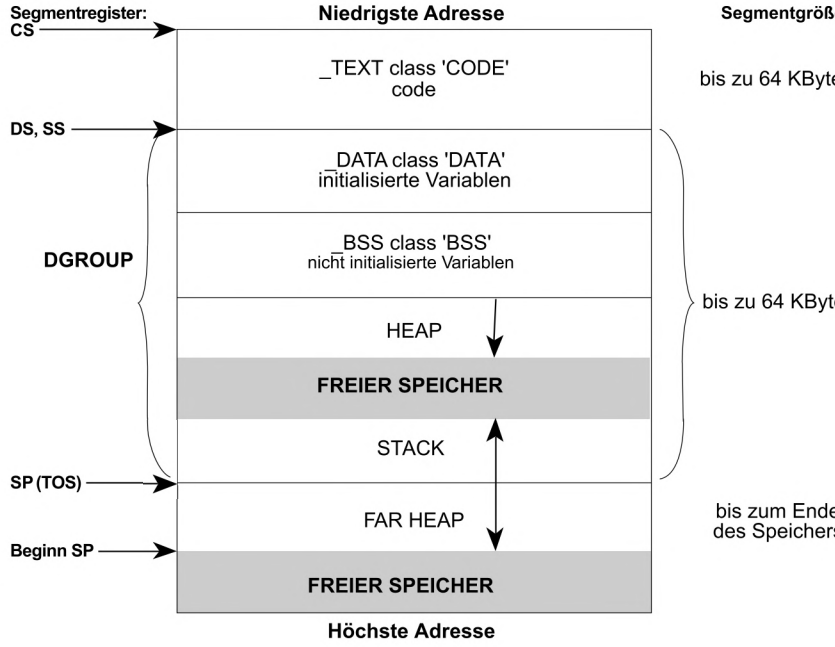


Abbildung 1.5:
Segmentierung im Modell Medium

CS kann immer nur auf ein sfile zeigen.

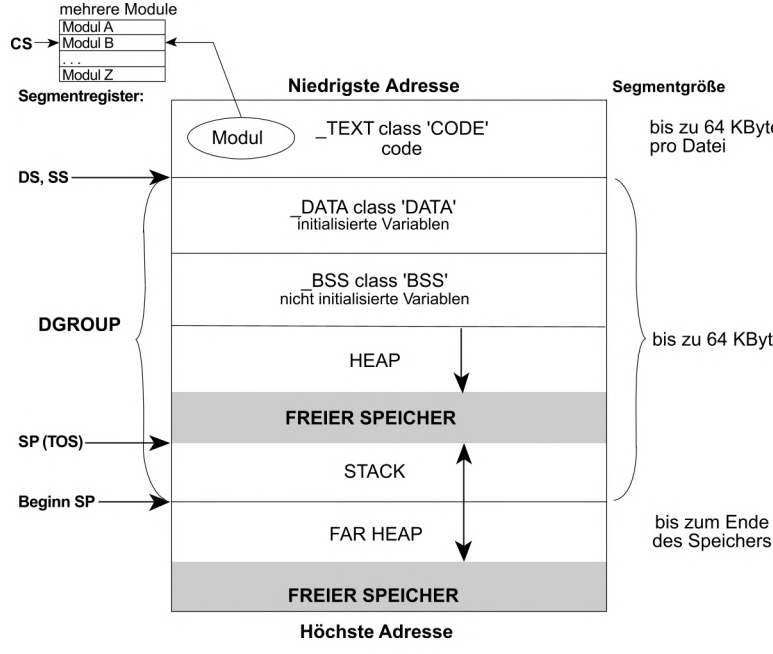


Abbildung 1.6: Segmentierung im Modell Compact

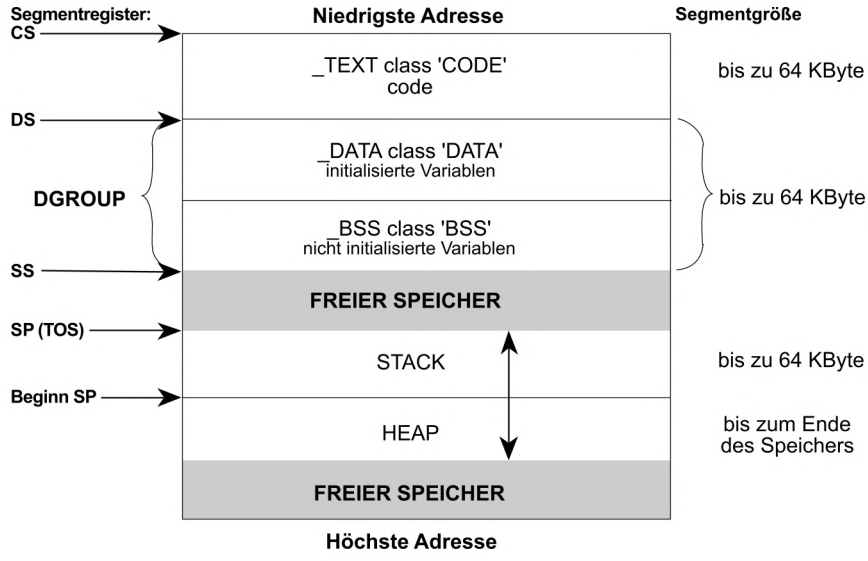


Abbildung 1.7: Segmentierung im Modell Large

CS kann immer nur auf ein sfile zeigen.

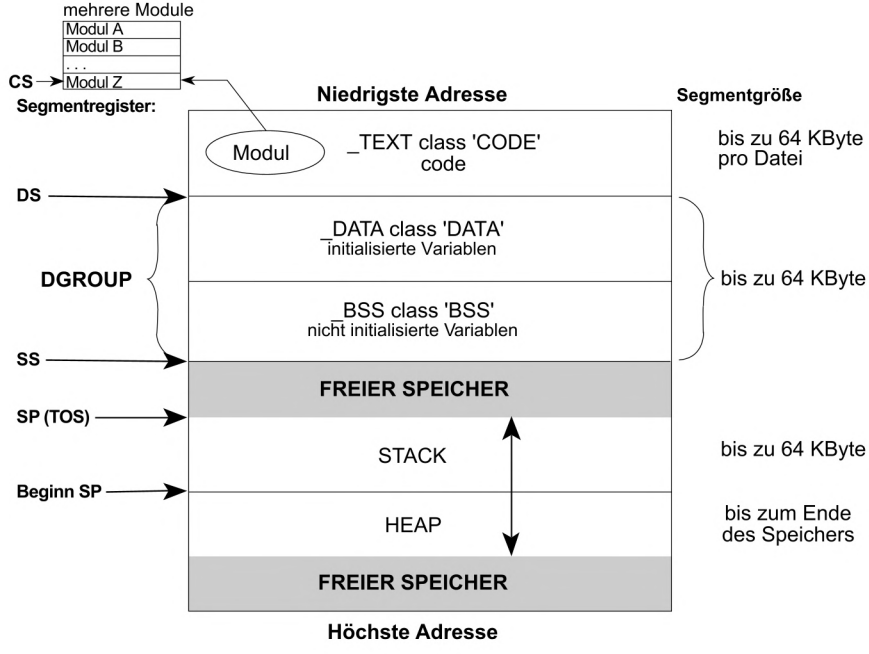


Abbildung 1.8: Segmentierung im Modell Huge

CS und DS können immer nur auf ein sfile zeigen.

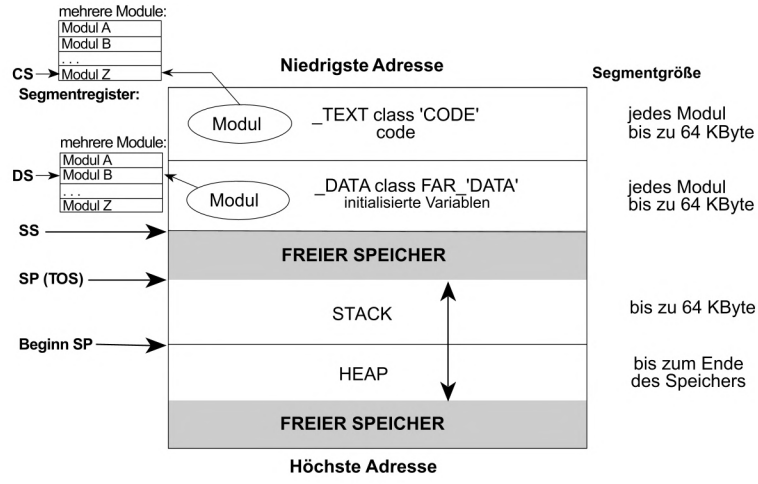


Tabelle 1.1 faßt die verschiedenen Speichermodelle zusammen und zeigt ihre Unterschiede hinsichtlich Code- und Datengröße auf. Aus der Tabelle ist ersichtlich, daß sich die Modelle in zwei Gruppen einteilen: *kleine* (64 KByte) und *große* (16 MByte) Modelle.

Tabelle 1.1: Speichermodelle

Die Modelle *Tiny*, *Small* und *Compact* werden im allgemeinen als »kleine Code-Modelle« bezeichnet, da hier maximal 64 KByte Programmcode möglich sind (und Zeiger auf Funktionen als **near** deklariert werden). Analog hierzu werden die Modelle *Compact*, *Large* und *Huge* als »große Datenmodelle« bezeichnet, weil hier standardmäßig der Datenzeigertyp **far** verwendet wird.

Datengröße	Code-Größe	
	16 KByte	**16 MByte / 4 GByte**
64 KByte	*Tiny* (Daten und Code überlappen; Gesamtgröße = 64 KByte) *Small* (Kein Überlappen; Gesamtgröße = 128 KByte)	*Medium* (wenige Daten, vieler Code)
16 MByte/ 4 GByte	*Compact* (viele Daten, weniger Code)	*Large* (viele Daten, vieler Code) *Huge* (wie *Large*, aber mit statischen Daten: 64 KByte)

Beim Compilieren eines Moduls (also eines Quelltextes zusammen mit eventuellen Include-Dateien) kann der erzeugte Code 64 KByte nicht überschreiten, da er in ein Code-Segment hinein passen muß. Das gilt auch dann, wenn Sie ein großes Code-Modell verwenden (*Medium*, *Large*

oder *Huge*). Ist ein Modul größer als 64 KByte, müssen Sie es in mehrere Quelltextdateien aufteilen, diese Dateien getrennt compilieren und sie dann miteinander linken. Dasselbe gilt für statische Variablen: Selbst im Modell *Huge* darf ihr Umfang innerhalb *jedes einzelnen* Moduls die 64-KByte-Grenze nicht überschreiten.

Das Programmieren mit gemischten Speichermodellen: Modifizierer

Borland C++ enthält über den ANSI-C-Standard hinaus noch acht weitere Modifizierer, die (unter Beachtung verschiedener Vorsichtsmaßnahmen) auf Zeiger und in manchen Fällen auch auf Funktionen angewendet werden können: **__near**, **__far**, **__huge**, **__cs**, **__ds**, **__es**, **__ss** und **__seg**.

Mit den Schlüsselwörtern **__near**, **__far** und **__huge** lassen sich die Deklarationen von Zeigern, Objekten und Funktionen modifizieren. Die Datenzeiger **__near**, **__far** und **__huge** wurden bereits in diesem Kapitel besprochen. Mit dem Schlüsselwort **__far** können Objekte als **__far** deklariert werden. Als **__near** deklarierte Funktionen werden mit **__near** aufgerufen (»near call«) und enden auch mit einem entsprechenden Rücksprung (»near return«). Die Deklaration einer Funktion als **__far** hat den Aufruf mit **__far** zur Folge, der Rücksprung erfolgt ebenfalls mit **__far**. Der Modifizierer **__huge** deklariert eine Funktion ebenfalls als **__far**, setzt aber – im Gegensatz zu **__far**-Funktionen – das DS-Register auf einen neuen Wert.

Bei den vier speziellen **__near**-Datenzeigern **__cs**, **__ds**, **__es** und **__ss** handelt es sich um 16-Bit-Zeiger, die auf das jeweilige Segmentregister festgelegt sind. Eine Deklaration wie

```
char __ss *p;
```

erzeugt also einen Zeiger *p*, der einen 16-Bit-Offset des Stacksegments enthält.

Welche Art von Funktionen und Zeigern Borland C++ erzeugt, hängt vom jeweils verwendeten Speichermodell ab. **__near**-Zeiger und -Funktionen benutzen automatisch die Register CS oder DS.

Tabelle 1.2 faßt zusammen, welche Zeigertypen Borland C++ bei welchem Speichermodell automatisch verwendet – ist der jeweilige Bereich größer als 64 KByte, wird far verwendet (far hat seine eigene Segmentadresse und einen Bereich von einem MByte). Ansonsten ist die Standardvorgabe der Typ near mit einem Bereich von 64 KByte:

Tabelle 1.2: Standard-Zeigertypen versus Speichermodelle

Speichermodell	Funktionszeiger	Datenzeiger
Tiny	near,_cs	near,_ds
Small	near,_cs	near,_ds
Medium	far	near,_ds
Compact	near,_cs	far
Large	far	far
Huge	far	far

Segment-Zeiger

Mit dem Modifizierer **__seg** können Sie 16-Bit-Segmentzeiger deklarieren. Die Syntax dafür lautet:

Datentyp _seg **Bezeichner*;

Zum Beispiel:

```
int _seg *name;
```

Bei jeder Indirektion über *Bezeichner* wird ein Offset von 0 unterstellt. Für Berechnungen mit Segmentzeigern gelten folgende Regeln:

1. Die Operatoren **++**, **—**, **+=** und **–=** dürfen nicht auf Segmentzeiger angewendet werden.
2. Die Differenz zwischen zwei Segmentzeigern darf nicht berechnet werden.
3. Aus der Addition eines Segmentzeigers und eines near-Zeigers resultiert ein far-Zeiger. Er ergibt sich aus dem Segment des Segmentzeigers und dem Offset des near-Zeigers. Beide Zeiger müssen daher auf denselben Typ zeigen, oder einer der beiden Zeiger muß ein Zeiger auf void sein. Der Offset wird nicht multipliziert, gleichgültig, auf welchen Typ er zeigt.
4. Wird ein Segmentzeiger innerhalb eines Indirektionsausdrucks benutzt, so wird er ebenfalls implizit in einen **far**-Zeiger konvertiert.
5. Wird ein Integer-Operand zu einem Segmentzeiger addiert bzw. von ihm subtrahiert, so ergibt sich ein **far**-Zeiger. Das Segment des **far**-Zeigers wird aus dem Segmentzeiger ermittelt. Der Offset wird ermittelt, indem die Größe des Objekts, auf das der Segmentzeiger zeigt, mit dem Integer-Operanden multipliziert wird. Die Berechnung findet genauso statt, als handele es sich bei dem Zeiger um einen **far**-Zeiger.
6. Segmentzeigern können Werte zugewiesen, initialisiert, von und zu Funktionen übergeben werden usw. Sie werden miteinander verglichen, als ob sie vom Typ **unsigned int** wären. Abgesehen von den genannten Einschränkungen werden sie also genauso behandelt wie jeder andere Zeiger.

Deklarieren von far-Objekten

In Borland C++ können Sie far-Objekte deklarieren. Zum Beispiel:

```
int far x = 5;
int far z;
extern int far y = 4;
static long j;
```

Für jedes far-Objekt wird ein eigenes Segment erzeugt.

Die Kommandozeilenoptionen **-zE**, **-zF**, und **-zH**, die auch mit der **#pragma option** gesetzt werden können, beziehen sich auf **far-**Segmentname, -klasse und -gruppe.

Diese Optionen, die Sie jederzeit ändern können, werden auf alle folgenden Deklarationen von far-Objekten angewendet. Das folgende Beispiel erzeugt ein far-Objekt in einem speziellen Segment:

```
#pragma option -zEmysegment -zHmygroup -zFmyclass
int far x;
#pragma option -zE* -zH*-zF*
```

Damit wird *x* in das Segment MYSEGMENT 'MYCLASS' in der Gruppe 'MYGROUP' plaziert. Anschließend werden die **far**-Objekt-Optionen auf ihre voreingestellten Werte zurückgesetzt. Mit Hilfe dieser Optionen können Sie sogar erzwingen, daß mehrere far-Objekte in einem einzigen Segment gespeichert werden:

```
#pragma option -zEcombined -zFmyclass
int far x;
double far y;
#pragma option -zE* -zF*
```

x und *y* erscheinen im Segment COMBINED 'MYCLASS' ohne Gruppe.

Explizite Funktionsdeklarationen mit near und far

In bestimmten Situationen kann es erforderlich sein, den durch das Speichermodell vorgegebenen Funktionstyp zu ändern.

Angenommen, Sie verwenden in einem Programm das Speichermodell *Large* und definieren diese rekursive Funktion:

```
double power(double x, int exp) {
   if (exp <= 0)
      return(1);
   else
      return(x * power(x, exp-1));
   }
```

Bei jeder Rekursion führt *power* einen far-Aufruf aus, was nicht nur Zeit, sondern auch die doppelte Menge an Platz auf dem Stack kostet. Durch

die Deklaration von *power* als **__near** wird dieser überhöhte Aufwand beseitigt:

```
double __near power(double x,int exp)
```

Nach einer derartigen Deklaration kann *power* nur innerhalb des Code-Segments aufgerufen werden, in dem diese Funktion compiliert wurde. Alle Aufrufe der Funktion sind near-Aufrufe.

Wenn Sie ein großes Modell verwenden (*Medium*, *Large* und *Huge*), kann die Funktion *power* nur aus dem Modul aufgerufen werden, in dem sie definiert ist. Da jedes Modul sein eigenes Code-Segment hat, ist der Aufruf von **__near**-Funktionen in anderen Modulen nicht möglich. Eine **near**-Funktion muß vor ihrem ersten Aufruf definiert oder deklariert worden sein, sonst erzeugt der Compiler einen **far**-Aufruf, während sie selbst einen near-Aufruf erwartet.

Umgekehrt wird bei der Deklaration einer Funktion als far ein far-Rücksprung vom Compiler erzeugt. Bei den kleinen Datenmodellen (*Tiny*, *Small* und *Medium*) muß die **far**-Funktion vor ihrem ersten Aufruf definiert oder deklariert werden, um sicherzustellen, daß sie mit einem **far**-Aufruf aufgerufen wird.

Sehen wir uns noch einmal das Beispiel mit *power* an. Es ist sinnvoll, *power* auch als **static** zu deklarieren. Auf diese Weise wird sichergestellt, daß Aufrufe nur innerhalb des aktuellen Moduls zulässig sind und der Name als Funktion außerhalb des Moduls nicht zur Verfügung steht.

Explizite Zeigerdeklarationen mit near, far und huge

Explizite Zeigerdeklarationen mit **__near**, **__far** und **__huge** haben natürlich dieselben Gründe wie entsprechende Funktionsdeklarationen (mit **near** und **far**): die Vermeidung unnötigen Ballasts (durch die Deklaration als **__near**, wenn **__far** vorgegeben ist) oder Zugriffe auf andere Teile des Hauptspeichers (durch die Deklaration als **__far** oder **__huge**, wenn **__near** vorgegeben ist).

Explizite Deklarationen – unabhängig davon, ob es sich um Funktionen oder Zeiger handelt, können aber auch Probleme mit sich bringen. Sehen wir uns dazu das folgende Beispiel für ein kleines Datenmodell an:

```
void myputs(s) {
   char *s;
   int i;
   for (i = 0; s[i] != 0; i++) putc(s[i]);
   }

main() {
   char near *mystr;
```

```
mystr = "Hello, world\n";
myputs(mystr);
}
```

Solange dieses Programm als Tiny oder Small compiliert wird, ergeben sich keine Probleme. Die Deklaration von *mystr* als **__near** ist eigentlich überflüssig, weil sowieso durchgehend mit near-Zeigern gearbeitet wird.

Aber was passiert, wenn Sie das Programm unter Verwendung der Speichermodelle Compact, Large oder Huge neu compilieren? Der Zeiger *mystr* bleibt in *main* als near deklariert (*mystr* ist weiterhin ein 16-Bit-Zeiger). Der Zeiger *s* in *myputs* ist aber jetzt (standardmäßig) **__far**. *myputs* liest nun aber 32 Bit vom Stack, um einen **__far**-Zeiger zu erzeugen, und erhält somit nicht die Adresse von *mystr*.

Dieses Problem läßt sich durch die Definition von *myputs* im modernen C-Stil vermeiden:

```
void myputs(char *s) {
   /* Rumpf von myputs */
   }
```

Explizite Deklarationen von Zeigern als **__far** oder **__near** sollten nur zusammen mit vollständigen Definitionen und Prototypen verwendet werden.

Bei der Compilierung des Programms weiß Borland C++, daß *myputs* einen Zeiger auf **char** erwartet.Da mit einem Large-Modell compiliert wird, ist außerdem bekannt, daß der Zeiger vom Typ **__far** sein muß. Borland C++ legt deshalb das DS-Register zusammen mit dem 16-Bit-Wert von *mystr* auf den Stack und bildet so einen **__far**-Zeiger.

Im umgekehrten Fall, bei einer Deklaration von *myputs* als **__far** und der Compilierung mit einem kleinen Datenmodell, gilt genau dasselbe – ohne Prototyp kommt es zu Problemen, da *main* den Offset und die Segmentadresse auf den Stack legt, *myputs* aber nur den Offset erwartet. Mit der Prototyp-Funktionsdefinition dagegen legt *main* nur den Offset auf den Stack.

Direktes Setzen von Zeigern (Segment:Offset-Adresse)

Wie läßt sich ein Far-Zeiger so setzen, daß er auf eine bestimmte Speicheradresse (eine spezielle Segment:Offset-Kombination) zeigt? Für derartige Operationen können Sie das Makro *MK_FP* einsetzen. *MK_FP* erwartet als Parameter das Segment und den Offset und liefert einen far-Zeiger zurück:

```
MK_FP(segment_value, offset_value)
```

Für den **__far**-Zeiger *fp* liefert *FP_SEG*(*fp*) die Segmentadresse und *FP_OFF*(*fp*) den Offset. Das *Referenzhandbuch* enthält ausführliche Informationen zu diesen Bibliotheksroutinen.

Bibliotheksdateien

Borland C++ enthält für jedes Speichermodell eine Datei mit Standardbibliotheksroutinen. Das Einbinden der Bibliotheken erfolgt selbständig, wobei je nach verwendetem Speichermodell die entsprechenden Dateinamen übergeben werden. Falls Sie mit der Kommandozeilenversion von Borland C++ arbeiten und den Linker TLINK direkt aufrufen wollen, müssen Sie die verwendeten Bibliotheken explizit angeben (siehe *Benutzerhandbuch*).

Linken von Modulen mit unterschiedlichen Speichermodellen

Das Linken von Modulen mit unterschiedlichen Speichermodellen ist unter Umständen mit Schwierigkeiten verbunden: Zwei Module, von denen eines zum Beispiel mit dem Modell Small, das andere aber mit Large compiliert wurde, verursachen keine weiteren Fehlermeldungen des Linkers. Bei Aufrufen von Funktionen des einen Moduls durch das andere treten aber dieselben Probleme auf, die bereits im Abschnitt »Explizite Funktionsdeklarationen mit near und far« besprochen wurden. Ruft z.B. eine Funktion im Small-Modul eine Funktion im Large-Modul auf, geschieht dies mit einem **near**-Aufruf, was wahrscheinlich zu einem unbrauchbaren Ergebnis führt. Die gleichen Probleme ergeben sich bei Zeigern (vgl. den Abschnitt »Explizite Zeigerdeklarationen mit near, far und huge«), da Funktionen im Small-Modul **__near**-Zeiger, Funktionen im Large-Modul dagegen **__far**-Zeiger erwarten.

Die Lösung dieser Probleme besteht wiederum in der Verwendung von Funktionsprototypen. Angenommen, die Funktion *myputs* wird in ihrem eigenen Modul mit dem Modell Large compiliert, und anschließend wird eine Header-Datei erzeugt (zum Beispiel myputs.h), die den folgenden Prototyp enthält:

```
void far myputs(char far *s);
```

Danach kann *main* in einem weiteren Modul (z.B. MYMAIN.C) geschrieben werden:

```
#include <stdio.h>
#include "myputs.h"

main() {
   char  near *mystr;

   mystr = "Hello, world\n";
   myputs(mystr);
   }
```

Bei der Compilierung dieses Textes liest Borland C++ den Prototyp von MYPUTS.H und stellt fest, daß die Funktion *myputs* als **__far** deklariert ist und einen **__far**-Zeiger erwartet. Für Aufrufe von *myputs* wird des-

halb der entsprechende Aufruf-Code erzeugt – unabhängig vom momentan verwendeten Speichermodell.

Ein Problem bleibt noch zu lösen: Wie werden Bibliotheksroutinen von den verschiedenen Modulen aus aufgerufen? Hier ist es am besten, eine Bibliothek für ein großes Speichermodell zu verwenden und alle Bibliotheksroutinen als **__far** zu deklarieren. Dazu muß von allen Deklarationsdateien, die normalerweise via *#include* in das Programm aufgenommen würden (zum Beispiel stdio.h), eine Kopie hergestellt und umbenannt werden (z.B. in fstdio.h).

Danach müssen Sie die Prototypen sämtlicher Funktionen und die dazugehörigen Parameter in der Kopie explizit als **__far** deklarieren, also zum Beispiel:

```
int far cdecl printf(char far * format, ...);
```

Auf diese Weise wird nicht nur dafür gesorgt, daß sämtliche Bibliotheksroutinen als **__far** aufgerufen werden – auch alle übergebenen Zeiger haben automatisch diesen Typ. Der letzte notwendige Schritt besteht in der Aufnahme der neuen Header-Datei via *#include* in die Quelltexte Ihrer Module:

```
#include <fstdio.h>
void main() {
   char  near *mystr;
   mystr = "Hello, world\n";
   printf(mystr);
}
```

Compilieren Sie Ihre Module einzeln mit dem Kommandozeilencompiler BCC und linken Sie danach sämtliche .OBJ-Dateien mit TLINK unter Verwendung einer großen Speichermodellbibliothek wie CL.LIB. Das Mischen von Modellen ist zwar recht schwierig, kann aber durchgeführt werden. Sie müssen jedoch mit schwer lokalisierbaren Fehlern rechnen, wenn Sie dabei Fehler machen.

Overlays (VROOM) für DOS

Overlays werden nur für 16-Bit-DOS-Programme verwendet. Sie können die Programmsegmente einer Windows-Anwendung mit dem Attribut *discardable* versehen, um den Speicherverbrauch zu reduzieren.

Overlays sind Teile eines Programms, die einen gemeinsamen Speicherbereich nutzen. Es werden immer nur diejenigen Programmteile im Spei-

cher gehalten, die für die gegenwärtige Funktion benötigt werden (vgl. *Benutzerhandbuch*).

Die zur Ausführung eines Programms erforderliche Speicherkapazität wird auf diese Weise beträchtlich reduziert. Somit können also auch Programme ausgeführt werden, die um einiges größer sind als der verfügbare Speicher.

Die Arbeitsweise von Overlays

Die Overlay-Verwaltung von Borland C++, auch VROOMM (Virtual Runtime Object-Oriented Memory Manager) genannt, nimmt Ihnen einen beträchtlichen Teil der Arbeit ab. In einem konventionellen Overlay-System werden Module in ein Basismodul und eine Menge von Overlay-Einheiten eingeteilt. Die Routinen in einer Overlay-Einheit können andere Routinen in derselben Einheit und in der Basis aufrufen, nicht aber solche aus anderen Einheiten. Die Overlay-Einheiten überlagern sich gegenseitig, das heißt, es befindet sich immer nur eine Overlay-Einheit im Speicher, und alle Einheiten belegen jeweils denselben physikalischen Speicherplatz. Der zur Ausführung des Programms erforderliche Gesamtbedarf an Speicherkapazität errechnet sich aus der Größe der Basis und der Größe des umfangreichsten Overlays.

Das konventionelle Overlay-System ist nicht besonders flexibel. Es erfordert die Kenntnis aller möglichen Aufrufabhängigkeiten im Programm und eine entsprechende Gruppierung der Overlays. Wenn eine Aufteilung des Programms in mehrere Aufrufabhängigkeiten nicht möglich ist, kann das die Einrichtung von Overlays verhindern.

Die Overlay-Verwaltung von Borland C++ unterscheidet sich in einigen Punkten vom üblichen System, indem sie eine *dynamische Segmentauslagerung* ermöglicht. Die grundlegende Auslagerungseinheit ist das Segment. Ein Segment kann aus einem oder mehreren Modulen bestehen und – was wichtiger ist – jedes Segment kann *jedes andere* Segment aufrufen.

Der Speicher wird in zwei Bereiche aufgeteilt: Basis- und den Auslagerungsbereich. Wird eine Funktion in einem Segment aufgerufen, das sich weder im Basis- noch im Auslagerungsbereich befindet, wird das entsprechende Segment in den Auslagerungsbereich gebracht. Dadurch werden eventuell andere Segmente ersetzt. Auf diese Weise wird eine Art virtuelle Speicher geschaffen, der die Aufteilung des Programms in einzelne statische Overlay-Einheiten überflüssig macht.

Was geschieht nun aber beim Laden eines Segments in den Auslagerungsbereich? Ist Platz für das Segment vorhanden, wird die Ausführung fortgesetzt; ist dies nicht der Fall, muß zumindest ein Segment aus dem Auslagerungsbereich entfernt werden.

Der Algorithmus für die Auswahl des auszulagernden Segments ist ziemlich kompliziert, entspricht aber im Prinzip der folgenden vereinfachten Darstellung: »Gibt es ein inaktives Segment, so lagere dieses aus (ein Segment ist inaktiv, wenn es für ausführende Funktionen aktuell nicht benötigt wird). Ist kein inaktives Segment vorhanden, so lagere ein aktives aus. Lagere so viele Segmente aus, bis genügend Platz verfügbar ist.« Diese Technik heißt *dynamische Auslagerung*.

Die Ausführungsgeschwindigkeit des Programms nimmt mit der Größe des zur Verfügung stehenden Auslagerungsbereiches zu. Der Swap-Bereich arbeitet in dieser Hinsicht wie ein Cache-Speicher: Je größer der Cache ist, desto schneller läuft das Programm. Die optimale Größe des Auslagerungsbereichs ist die Größe des *arbeitenden* Teils des Programms.

Nach dem Laden eines Overlays in den Speicher wird es in dem Overlay-Puffer plaziert. Dieser befindet sich zwischen dem Stack-Segment und dem »Far-Heap«. Die Größe des Overlay-Puffers wird standardmäßig beim Start des Programms geschätzt und festgelegt, kann aber mit der globalen Variablen *_ovrbuffer* geändert werden (vgl. Kapitel 4 im *Referenzhandbuch*). Ist nicht genügend Speicher vorhanden, zeigen DOS oder der Startup-Code von C eine Fehlermeldung an:

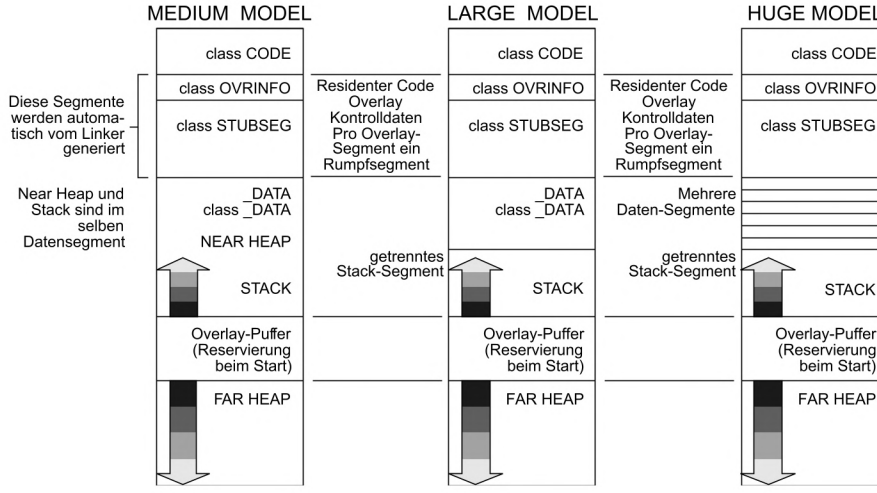
```
Program too big to fit in memory (DOS)
```

oder

```
Not enough memory to run program (C).
```

Eine wichtige Eigenschaft der Overlay-Verwaltung ist die Fähigkeit, aus dem Overlay-Puffer ausgelagerte Module im *Expanded Memory* oder *Extended Memory* zu speichern. Wird das Modul später wieder benötigt, kann es von dort wieder in den Overlay-Puffer eingelagert werden, das heißt, ein erneutes Lesen aus der Datei ist nicht erforderlich. Die Ausführungsgeschwindigkeit wird dadurch beträchtlich erhöht. Abbildung 1.9 zeigt, wie der Speicher bei der Verwendung von Overlays genutzt wird.

Abbildung 1.9: Die Belegung des Speichers durch Overlays



Die optimale Nutzung von Borland C++ Overlays

Zur optimalen Ausnutzung des Overlay-Systems von Borland C++ sollten Sie folgendermaßen vorgehen:

- Minimieren Sie den RAM-residenten Code, (residente Laufzeitbibliothek, Interrupt-Handler, Gerätetreiber usw.).
- Legen Sie eine konfortable Pool-Größe für die Overlays fest – beginnen Sie mit 128 KByte, und versuchen Sie, durch Variieren der Größe (nach oben und unten), Geschwindigkeit und Größe zu optimieren.
- Setzen Sie das Overlay-System für spezielle Zwecke ein, z.B. für ein interaktives Hilfesystem, das Sie nicht von Anfang an berücksichtigt haben.

Voraussetzungen für die Erzeugung von Overlays

Siehe S. 29 für weitere Informationen über die Einstellung der Größe des Overlay-Puffers.

- Ein Segment ist der kleinste Programmteil, der als Overlay-Einheit eingesetzt werden kann.
- Das Overlay-System kann nur von Programmen genutzt werden, die eines der folgenden Speichermodelle verwenden: *Medium*, *Large* oder *Huge*. Die Modelle *Tiny*, *Small* und *Compact* werden nicht unterstützt.
- Overlay-Segmente sind den üblichen Regeln für die Segmentmischung unterworfen, das heißt, mehrere .OBJ-Module können zu demselben Overlay-Segment beitragen.

Die Erzeugung der Overlays durch den Linker ist von der Overlay-Verwaltung während der Laufzeit völlig unabhängig. Der Linker fügt *nicht* automatisch Code zur Verwaltung der Overlays ein. Vom Standpunkt des Linkers aus ist die Overlay-Verwaltung nur ein weiterer Bestandteil des Codes, der gelinkt werden muß. Er nimmt lediglich an, daß die Speicherverwaltung einen Interrupt-Vektor (normalerweise INT 3FH) umdefiniert, über den die dynamischen Ein- und Auslagerungsvorgänge gesteuert werden. Aufgrund dieser Transparenz kann auch auf einfache Weise eine eigene Overlay-Verwaltung implementiert werden, die den speziellen Anforderungen jeder Anwendung gerecht wird.

Overlays und Exception-Behandlung

Bei der Erstellung eines C++ Programms nach der Overlay-Technik mit Exception-Behandlung gibt es eine Reihe von Situationen, die Sie vermeiden müssen. Folgende Programmelemente dürfen keine Konstrukte zur Exception-Behandlung enthalten:

- Inline-Funktionen ohne Inline-Erweiterung
- Template-Funktionen
- Elementfunktionen von Template-Klassen

Die Konstrukte der Exception-Behandlung enthalten **try**/**catch**- und **__try**/**__except**-Blöcke. Zusätzlich kann der Compiler Exception-Behandlungsoperationen für Blöcke mit automatischen Klassenvariablen, Exception-Spezifikationen sowie einigen **new**/**delete**-Ausdrücken einfügen.

Falls Sie eines dieser Konstrukte in der Overlay-Technik zu verwenden versuchen, wird der Linker das Modul und die Funktion in folgender Weise ausfindig machen:

```
Error: Illegal local public in Functionsname in module Modulname
```

Falls dieser Fehler durch eine Inline-Funktion hervorgerufen wird, schreiben Sie diese Funktion so um, daß sie nicht die *inline*-Option entält. Falls der Fehler jedoch durch eine Template-Funktion ausgelöst wurde können Sie folgendes tun:

- Entfernen Sie alle Exception-Behandlungskonstrukte aus der Funktion.
- Entfernen Sie die Funktion aus dem Overlay-Modul.

Bei der Mehrfachvererbung in der Overlay-Technik müssen Sie besonders vorsichtig sein. Der Versuch, ein Modul zu überlagern, das Klassenkonstruktoren oder -destruktoren für Mehrfachvererbug benutzt, kann den Linker dazu veranlassen, die Meldung

```
Error: Illegal local public in Klassenname:: in module Modulname
```

zu erzeugen. In diesem Fall sollte das Modul *Modulname* nicht überlagert werden.

Bei Containerklassen (in BIDS?.LIB) ist der Mechanismus für die Exception-Behandlung standardmäßig ausgeschaltet. Dennoch werfen die Diagnose-Versionen von BIDS-Bibliotheken Exceptions aus, weswegen sie nicht mit Overlays benutzt werden sollten. Aus dem gleichen Grund sollten auch keine *string*-Klassen benutzt werden. Weitere Informationen über BIDS in Verbindung mit *string*-Klassen finden Sie im *Referenzhandbuch*.

Verwenden von Overlays

Um Overlays in einem Programm zu verwenden, müssen dessen Module sämtlich mit der Compileroption **-Y** compiliert werden. Zur Übernahme eines bestimmten Moduls in ein Overlay muß dieses mit der Option **-Yo** compiliert werden (mit **-Yo** wird **-Y** automatisch aktiviert).

Overlays können nur in 16-Bit-DOS-Programmen eingesetzt werden.

Die Option **-Yo** gilt für alle Module und Bibliotheken, die in der Kommandozeile des Kommandozeilencompilers enthalten sind. Sie kann mit **-Yo-** wieder deaktiviert werden. Dies sind die einzigen Optionen, die einem Dateinamen folgen dürfen. Um zum Beispiel das Modul OVL.C, nicht aber die Bibliothek GRAPHICS.LIB zu einem Overlay zu machen, kann eine der folgenden Kommandozeilen benutzt werden:

```
BCC -ml -Yo ovl.c -Yo- graphics.lib
```

oder

```
BCC -ml graphics.lib -Yo ovl.c
```

Wird zum Linken der .EXE-Datei explizit TLINK aufgerufen, muß in der Linker-Kommandozeile oder in der Response-Datei die Linker-Option **/o** angegeben werden. Eine ausführliche Erläuterung von TLINK und der Option **/o** finden Sie im *Benutzerhandbuch*.

Ein Beispiel für ein Overlay

Angenommen, Sie wollen ein Programm mit Overlays erzeugen, das aus folgenden drei Modulen besteht: MAIN.C, O1.C und O2.C. Es sollen aber nur die Module O1.C und O2.C zu Overlays gemacht werden, da MAIN.C zeitkritische Routinen und Interrupt-Handler enthält. Gehen wir davon aus, daß das Speichermodell Large verwendet wird.

Weitere Informationen über Programmierung mit Overlays erhalten Sie im *Benutzerhandbuch*.

Das folgende Kommando führt die Aufgabe aus:

```
a BCC -ml -Y main.c -Yo o1.c o2.c
```

Das Resultat ist die ausführbare Datei MAIN.EXE, die zwei Overlays enthält.

Gestalten von Programmen mit Overlays

In diesem Abschnitt wird auf wichtige Punkte eingegangen, die bei der Erstellung von Programmen mit Overlays zu beachten sind.

Nur Far-Aufrufe von Funktionen!

Beim Compilieren eines Overlay-Moduls muß ein großes Speichermodell verwendet werden (*Medium*, *Large* oder *Huge*). Des weiteren *muß* bei jedem Aufruf einer Overlay-Funktion in einem anderen Modul sichergestellt werden, daß alle momentan aktiven Funktionen far-Funktionen sind.

Alle Overlay-Module *müssen* mit der Option **-Y** compiliert werden. Damit wird sichergestellt, daß der erzeugte Code als Overlay eingesetzt werden kann.

Ist dies nicht der Fall, so kann es bei der Ausführung des Programms zu unvorhergesehenen – eventuell sogar zu katastrophalen – Auswirkungen kommen.

Die Größe des Overlay-Puffers

Der Overlay-Puffer hat standardmäßig die doppelte Größe des umfangreichsten Overlays – dies ist für viele Programme angemessen. Wird jedoch eine bestimmte Funktion des Programms von vielen Overlay-Modulen benötigt und übersteigt die Gesamtgröße dieser Module den Umfang des Overlay-Puffers, so hat dies bei häufigen gegenseitigen Aufrufen sehr viele Ein- und Auslagerungsvorgänge zur Folge.

Als Lösung bietet sich an, die Größe des Overlay-Puffers zu erhöhen, so daß jederzeit genügend Speicher für alle Overlays vorhanden ist. Die Vergrößerung kann durch Setzen der globalen Variablen *_ovrbuffer* auf die gewünschte Größe in Paragraphen erreicht werden.

Mit der folgenden Anweisung kann z.B. die Größe des Overlay-Puffers auf 128 KByte gesetzt werden:

```
unsigned _ovrbuffer = 0x2000;
```

Es gibt keine allgemeingültige Formel zur Bestimmung der idealen Größe des Overlay-Puffers.

Wann sollten Overlays nicht eingesetzt werden?

Exception-Behandlungskonstrukte in Overlays erfordern besondere Aufmerksamkeit. Auf Seite 33 wird das Verfahren genauer beschrieben.

Module mit Interrupt-Handlern oder kleinen zeitkritischen Routinen sollten nicht zu Overlays gemacht werden. Da DOS nicht re-entrant ist, gilt dies auch für Module, die von Interrupt-Funktionen aufgerufen werden.

Die Overlay-Verwaltung von Borland C++ unterstützt die Übergabe von Overlay-Funktionen als Argumente, die Zuweisung und Initialisierung

von Funktionszeigervariablen mit Adressen von Overlay-Funktionen und den Aufruf von Overlay-Routinen über Funktionszeiger.

Debuggen von Overlays

Overlays sollten nicht mit einer diagnostischen Version der BIDS-Bibliotheken verwendet werden.

Die meisten Debugger können Overlays nur sehr eingeschränkt (oder gar
nicht) austesten. Dies ist bei dem Turbo Debugger nicht der Fall. Er unterstützt auf eine für den Programmierer transparente Weise die schrittweise Abarbeitung von Overlays, in denen auch Haltepunkte gesetzt werden können. Mit Hilfe von Overlays können somit sehr große Programme einfach bearbeitet und ausgetestet werden.

Externe Assembler-Routinen in Overlays

Externe Routinen in Assembler müssen – ebenso wie die normalen C-Funktionen – bestimmte Voraussetzungen erfüllen, damit sie von der Overlay-Verwaltung korrekt verarbeitet werden können.

Wenn eine Assembler-Routine eine Overlay-Funktion aufrufen soll, *muß* sie als *FAR* deklariert werden *und* außerdem mit dem BP-Register einen Stack-Frame einrichten. Dazu ein Beispiel: *OtherFunc* ist eine Overlay-Funktion in einem anderen Modul und wird von *ExternFunc*, einer Routine in Assembler, aufgerufen. *ExternFunc* muß dazu vom Typ *FAR* sein und einen Stack-Frame einrichten:

```
ExternFunc      PROC    FAR
        push    bp              ;BP retten
        mov     bp,sp           ;Stack-Frame einrichten
        sub     sp,LocalSize    ;Speicherplatz für lokale
                                ;Variablen reservieren
        ...
        call    OtherFunc       ;Aufruf eines anderen
                                ;Overlay-Moduls
        ...
        mov     sp,bp           ;Speicher für lokale
                                ;Variablen freigeben
        pop     bp              ;BP wiederherstellen
        RET                     ;Zurück

ExternFunc      ENDP
```

In diesem Beispiel ist *LocalSize* die Größe der lokalen Variablen. Ist *LocalSize* Null, können die beiden Zeilen für die Reservierung und die Freigabe der lokalen Variablen weggelassen werden. Der BP-Stack muß aber auf jeden Fall eingerichtet werden, auch dann, wenn es keine Argumente oder Variablen auf dem Stack gibt.

Die oben genannten Voraussetzungen gelten auch dann, wenn *ExternFunc* nur einen indirekten Bezug auf Overlay-Funktionen enthält. Ruft

zum Beispiel *OtherFunc* Overlay-Funktionen auf, ohne selbst eine zu sein, muß *ExternFunc* trotzdem *FAR* sein und einen Stack-Frame einrichten.

Falls die Assembler-Routinen keine direkten oder indirekten Bezüge auf Overlay-Funktionen enthalten, gibt es keine besonderen Anforderungen. In diesem Fall können die Assembler-Routinen als *NEAR* deklariert werden und es braucht kein Stack-Segment eingerichtet zu werden.

Overlay-Routinen in Maschinensprache sollten *keine* Variablen im Code-Segment erzeugen, da alle Modifikationen eines Overlay-Code-Segments bei der Auslagerung des Overlays verlorengehen. Das gleiche gilt für Zeiger auf Objekte in Overlay-Code-Segmenten: Sie könnten ihre Gültigkeit nach Aufrufen anderer Overlays verlieren, da die Overlay-Code-Segmente von der Overlay-Verwaltung »nach eigenem Ermessen« ausgelagert werden.

Aus- und Einlagerung

Wenn Sie *Expanded Memory* oder *Extended Memory* zur Verfügung haben, können Sie die Overlay-Verwaltung anweisen, diesen Speicherbereich zu nutzen. Er dient dann zur Aufnahme von Modulen, die aus Platzgründen aus dem Overlay-Puffer ausgelagert werden müssen. Dies vergrößert die Geschwindigkeit beträchtlich, da die Module bei einem erneuten Bedarf nicht von der Platte eingelesen werden müssen.

Die Overlay-Speicherverwaltung ist in der Lage, den vorhandenen *Expanded/Extendend Memory* selbst zu finden und zu nutzen. Ihr kann aber auch ein festgelegter Bereich zugewiesen werden. Vorhandener *Extended Memory* wird allerdings nicht immer erfolgreich erkannt, da verschiedene Cache- und RAM-Disk-Programme den *Extended Memory* ohne Markierung übernehmen. Sie können dieses Problem vermeiden, indem Sie der Overlay-Verwaltung die Startadresse des *Extended Memory* mitteilen und den Umfang angeben, der problemlos genutzt werden kann.

Borland C++ hält zwei Funktionen bereit, die es Ihnen erlauben, *Expanded* und *Extended Memory* zu initialisieren. Im Kapitel über Bibliotheksfunktionen im *Referenzhandbuch* finden Sie dazu die Beschreibung der Funktionen *_OvrInitEms* und *_OvrInitExt*.

K A P I T E L 2

Arithmetik

In diesem Kapitel werden die Gleitkomma-Optionen und der Gebrauch der numerischen Typen *bcd* und *complex* erläutert.

Ein-/Ausgabe von Gleitkommawerten

Für die Ausgabe von Gleitkommawerten müssen die von den Funktionen *printf* und *scanf* (sowie aller verwandten Funktionen) verwendeten Konvertierungsroutinen hinzugelinkt werden. Diese werden nicht automatisch hinzugelinkt, um das ausführbare Programm nicht unnötig zu vergrößern. Wenn Ihr Programm jedoch Gleitkommaroutinen oder Adressen von Gleitkommazahlen verwendet, so werden diese Routinen automatisch hinzugelinkt. Ist dies nicht der Fall, so können die fehlenden Konvertierungsroutinen zu einem Laufzeitfehler führen.

Das folgende Beispiel verdeutlicht, wie Sie in Ihrem Programm Gleitkomma-Ausgaben vorbereiten können:

```
/* Vorbereitung von Gleitkomma-Ausgaben */
#include <stdio.h>

#pragma extref _floatconvert

void main() {
  printf("d = %f\n", 1.3)
}
```

Gleitkomma-Optionen

In C werden zwei Arten von Zahlen unterschieden: Integerzahlen (**int**, **short**, **long** usw.) und Gleitkommazahlen (**float**, **double** und **long double**).

Der Prozessor Ihres Rechners ist für eine effektive Verarbeitung von Integerwerten eingestellt. Die Verarbeitung von Gleitkommawerten erfordert jedoch mehr Zeit und Aufwand.

Zur Prozessorfamilie iAPx86 gehört eine entsprechende Familie von mathematischen Koprozessoren, die 8087-, 80287- und 80387-Prozessoren. Wir bezeichnen die gesamte Prozessorfamilie als 80x87 oder einfach als »Koprozessor«.

Wenn Sie einen 80486 oder Pentium als Prozessor haben, ist der numerische Koprozessor wahrscheinlich bereits integriert.

Der 80x87 ist ein mathematischer Prozessor, der in Ihren PC eingebaut werden kann. Er führt Gleitkommabefehle sehr schnell aus. Wenn Sie sehr viel mit Gleitkommabefehlen arbeiten, werden Sie wahrscheinlich einen Koprozessor haben wollen. Die CPU in Ihrem Computer ist über spezielle Hardware-Leitungen mit dem 80x87 verbunden.

Emulation des 80x87

In Borland C++ ist *Emulation* (der Parameter **-f** des Kommandozeilencompilers) für die Option *Code-Generierung* voreingestellt. Diese Option kann bei Programmen eingesetzt werden, die mit oder ohne Gleitkommazahlen arbeiten und auf Computern mit oder ohne mathematischen Koprozessoren laufen sollen.

Mit der Option *Emulation* erzeugt der Compiler ein Programm so, als ob ein Koprozessor vorhanden wäre, linkt aber auch die Emulationsbibliothek (EMU.LIB) dazu. Bei der Ausführung des Programms wird der Koprozessor verwendet, falls einer vorhanden ist; steht zur Laufzeit kein Koprozessor zur Verfügung, setzt das Programm spezielle Routinen ein, die den 80x87 emulieren. Diese Routinen brauchen 512 Bytes vom Stack. Daher ist es notwendig, bei der Wahl der Emulationsoption, die Stackgröße entsprechend zu wählen.

Verwendung des 80x87-Codes

Wenn Ihr Programm nur auf einem Computer laufen soll, der mit einem 80x87-Koprozessor ausgestattet ist, können Sie in Ihren .EXE-Dateien bis zu 10 KBytes einsparen, weil dann die Überprüfung auf Vorhandensein eines 80x87 und die Emulation des 80x87 entfällt. Wählen Sie einfach die entsprechende Option im Dialogfenster *Compiler | Gleitkomma* (bzw. die Kommandozeilenoption **-f87**). Borland C++ linkt dann Ihre Programme mit FP87.LIB anstatt mit EMU.LIB.

Kein Gleitkomma-Code

Wenn Ihr Programm keinen Gleitkomma-Code enthält, können Sie etwas Zeit beim Linken einsparen, indem Sie die entsprechende Option im Dialogfenster *Compiler | Gleitkomma* wählen bzw. die Kommandozeilenoption **-f-** verwenden. In diesem Fall führt Borland C++ den Linkvorgang ohne die Bibliotheken EMU.LIB, FP87.LIB oder MATH*x*.LIB durch.

Schnelle Gleitkomma-Operationen

Borland C++ verfügt über eine Option zur Beschleunigung von Gleitkomma-Operationen (Kommandozeilenoption **-ff**). Sie kann in der Kommandozeile mit **-ff-** ausgeschaltet werden. Mit Hilfe dieser Option können gewisse Optimierungen durchgeführt werden, die im Gegensatz zur korrekten C-Semantik stehen. Dazu ein Beispiel:

```
double x;
_x = (float) (3.5*x);
```

Bei einer korrekten Ausführung wird *x* mit 3,5 multipliziert. Der resultierende **double**-Wert wird in einen **float**-Wert konvertiert und anschließend als **double**-Wert in *x* gespeichert. Beim Einsatz der entsprechenden Option wird das **long double**-Produkt direkt in einen **double**-Typ umgewandelt. Diese Option ist in Borland C++ voreingestellt, da die Korrektheit nur weniger Programme vom Genauigkeitsverlust abhängt, der sich aus der Zuweisung in einen kleineren Datentyp (**float**) ergibt.

Die Umgebungsvariable 87

Wenn Sie Ihr Programm mit der 80x87-Emulation linken (dies ist per Voreinstellung der Fall), überprüft das Programm automatisch, ob ein 80x87-Koprozessor zur Verfügung steht. Ist ein derartiger Prozessor vorhanden, wird er vom Programm verwendet.

Es gibt aber Situationen, in denen Sie diese voreingestellte Überprüfung ausschalten möchten: Nehmen wir z.B. an, daß Ihr eigenes Laufzeitsystem einen 80x87 hat, Sie aber überprüfen wollen, ob das Programm auch ohne Koprozessor korrekt arbeitet. Oder Ihr Programm soll auf einem PC-kompatiblen System laufen, welches falsche Informationen bei der Überprüfung auf Vorhandensein eines mathematischen Koprozessors liefert (z.B. daß ein nicht vorhandener 80x87 verfügbar ist oder umgekehrt).

Borland C++ stellt eine Option zur Verfügung, die die voreingestellte Logik der automatischen Überprüfung im Startup-Code außer Kraft setzt. Dabei handelt es sich um die Umgebungsvariable 87. Diese Variable setzen Sie nach dem DOS-Prompt mit dem SET-Kommando:

```
C> SET 87=N
```

oder

```
C> SET 87=Y
```

Vor und nach dem Gleichheitszeichen darf kein Leerzeichen stehen. Wenn Sie die Umgebungsvariable auf *N* (für *No*) setzen, wird dem Startup-Code mitgeteilt, daß Sie – auch bei Vorhandensein – keinen mathematischen Koprozessor verwenden wollen.

Wird die Variable auf *Y* (für *Yes*) gesetzt, bedeutet dies, daß der Koprozessor vorhanden ist und daß das Programm ihn verwenden wird. Seien Sie aber vorsichtig! Wenn Sie *87=Y* angeben und kein Koprozessor verfügbar ist, stürzt Ihr System ab.

Wurde die Umgebungsvariable definiert (mit einem beliebigen Wert), können Sie die Definition mit folgendem Kommando nach dem DOS-Prompt wieder aufheben:

```
C> SET 87=
```

Drücken Sie nach der Eingabe des Gleichheitszeichens die Eingabetaste.

Register und Koprozessor

Wenn Sie Gleitkommazahlen benutzen, sollten Sie sich im Zusammenhang mit Registern über zwei Punkte bewußt sein:

- Im 80x87-Emulationsmodus werden »wraparound« von Registern und bestimmte andere Eigenschaften des 80x87 nicht unterstützt.
- Wenn Sie gleichzeitig Gleitkommazahlen und Inline-Assemblercode verwenden, müssen Sie beim Ansprechen der 80x87-Registern eventuell besondere Vorsicht walten lassen. Mitunter ist es notwendig, die Registerinhalte des 80x87 zu sichern, bevor Funktionen aufgerufen werden, die den Koprozessor verwenden. Es sei denn, Sie sind sicher, daß genügend freie Register existieren.

Fehlerbedingungen bei Gleitkomma-Operationen

C++ Programme werden per Voreinstellung abgebrochen, wenn ein Gleitkomma-Überlauf oder eine Division durch Null auftreten.

Derartige Fehlerbedingungen können durch einen Aufruf von *_control87* in *main* vor der Ausführung von Gleitkomma-Operationen abgefangen werden. Betrachten Sie folgendes Beispiel:

```
#include <float.h>
main() {
   _control87 (MCW_EM,MCW_EM);
      ⋮
}
```

Durch den Aufruf von *_status87* oder *_clear87* können Sie feststellen, ob eine Fehlerbedingung aufgetreten ist. Eine detaillierte Erläuterung dieser Funktionen finden Sie im *Referenzhandbuch*.

Bestimmte arithmetische Fehler können auch in Bibliotheksfunktionen vorkommen, z.B. Quadratwurzel aus einer negativen Zahl. Normalerweise hat das eine Fehlermeldung zur Folge, und es wird NAN (not-a-number) zurückgeliefert. Durch die Verwendung von NAN ergibt sich später höchstwahrscheinlich eine Gleitkomma-Fehlerbedingung, die zum Pro-

grammabbruch führt, wenn sie nicht abgefangen wird. Sie können die Ausgabe der Fehlermeldung verhindern, indem Sie folgende *_matherr*-Version in Ihr Programm einfügen:

```
#include <math.h>
int _matherr(struct exception *e)
{
   return 1;            /* Der Fehler wurde abgefangen */
}
```

Der Einsatz von *_matherr* zur Behandlung anderer arithmetischer Fehler wird nicht empfohlen, da er nicht mehr gebräuchlich ist und in späteren Versionen von Borland C++ möglicherweise nicht mehr unterstützt wird.

Komplexe Arithmetik

Komplexe Zahlen sind Zahlen der Form $x + yi$, wobei x und y reelle Zahlen sind, und i die imaginäre Quadratwurzel von -1 ist. In Borland C ++ ist folgender Typ in math.h definiert:

```
struct complex
{
   double x, y;
};
```

Dieser Typ eignet sich zur Aufnahme komplexer Zahlen, die als Paare von reellen Zahlen betrachtet werden können. In C ist die Arithmetik mit komplexen Zahlen sehr mühsam – in C++ ist dies wesentlich einfacher.

Ein Vorteil des Borland C++ *complex*-Datentyps ist, daß er von allen mathematischen Routinen im ANSI C Standard benutzt werden kann. Der C Datentyp **struct complex** wird normalerweise vom ANSI C Standard nicht unterstüzt.

Eine ausführliche Beschreibung der Klasse *complex* finden Sie im Kapitel 8 des *Referenzhandbuchs*.

Wenn Sie in C++ mit komplexen Zahlen arbeiten möchten, müssen Sie dazu nur complex.h – mit Hilfe von *include* – einbinden. In complex.h sind folgende Operatoren und Funktionen überladen:

- Die binären arithmetischen Operatoren,
- Die Stream-Operatoren **>>** und **<<**,
- Die normalen ANSI-C Mathematikfunktionen.

Die *complex*-Bibliothek wird nur dann benuzt, wenn das Argument vom Typ *complex* ist. Die komplexe Quadratwurzel von -1 erhalten Sie daher mit

```
sqrt(complex(-1))
```

und nicht mit

```
sqrt(-1)
```

Die folgenden Funktionen sind in der Klasse *complex* definiert:

```
double  arg(complex&);   // Winkel in der komplexen Ebene
complex conj(complex&);  // konjugiert-komplexe Zahl
double  imag(complex&);  // Imaginärteil
double  norm(complex&);  // Quadrat der Betrags
double  real(complex&);  // Realteil
//Komplexe Zahl aus den Polarkoordinaten erzeugen
complex polar(double mag, double angle=0);
```

BCD-Arithmetik

Im Kapitel 8 des *Referenzhandbuchs* finden Sie weitere Informationen.

Die Arithmetik von Borland C++ arbeitet (wie auch fast alle Computer und Compiler) mit binären Zahlen (Basis 2). Dies ist manchmal verwirrend, wenn Sie nur die dezimale Darstellung (Basis 10) gewöhnt sind. Viele Zahlen, die im Dezimalsystem (Basis 10) exakt dargestellt werden, können mit binären Zahlen nur annähernd wiedergegeben werden (zum Beispiel 0.01).

Binäre Zahlen sind bei den meisten Anwendungen vorzuziehen, aber in einigen Fällen ist der Rundungsfehler, der bei der Umwandlung von Dezimal- in Binärzahlen entsteht, nicht tolerierbar. Dies trifft vor allem bei Finanz- oder Lohnbuchhaltungsprogrammen zu. Ein Beispiel soll dies verdeutlichen: das folgende Programm addiert 100 Pfennige auf und subtrahiert dann eine Mark:

```
#include <stdio.h>
int i;
float x = 0.0;
for (i = 0; i < 100; ++i)
   x += 0.01;
x -= 1.0;
printf("100*.01 - 1 = %g\n",x);
```

Das korrekte Ergebnis sollte 0.0 sein. Das Programm berechnet aber eine sehr kleine Zahl, die nur annähernd 0.0 ist. Die Berechnung vergrößert den winzigen Rundungsfehler, der bei der Umwandlung von 0.01 zur Basis 2 entsteht. Wird der Typ von *x* in den Typ **double** oder **long double** geändert, reduziert sich zwar der Fehler, er ist aber nicht eliminiert.

Zur Lösung dieses Problems bietet sich der C++ Typ *bcd* an, der in bcd.h deklariert ist. Mit *bcd* wird die Zahl 0.01 korrekt repräsentiert, und die *bcd*-Variable *x* liefert den korrekten Pfennigbetrag:

```
#include <bcd.h>
int i;
bcd x = 0.0;
for (i = 0; i < 100; ++i)
   x += 0.01;
x -= 1.0;
cout << "100 * .01 -1 = " << x << "\n";
```

Folgendes sollte beim Einsatz von *bcd* beachtet werden:

- *bcd* eliminiert nicht alle Rundungsfehler; bei Berechnungen wie 1.0/3.0 bleibt ein Rundungsfehler bestehen.
- Die gebräuchlichen math-Funktionen sind für *bcd*-Argumente überladen.
- bcd-Zahlen haben eine Genauigkeit von ca. 17 Dezimalstellen und umfassen einen Bereich zwischen 1×10^{-125} und 1×10^{125}.

Konvertierung von BCD-Zahlen

bcd ist ein vom Typ **float**, **double** oder **long double** verschiedener Datentyp; BCD-Arithmetik wird durchgeführt, wenn mindestens ein Operand vom Typ *bcd* ist.

Mit der *bcd*-Elementfunktion *real* kann eine *bcd*-Zahl in das Format zur Basis 2 (**float**, **double** oder **long double**) zurückkonvertiert werden. Die Konvertierung erfolgt allerdings nicht automatisch. *real* sorgt für die Konvertierung in den Typ **long double**, der dann mit den üblichen C-Konvertierungen in einen anderen Typ konvertiert werden kann. Zum Beispiel kann eine *bcd*-Zahl mit allen vier folgenden Ausgabeanweisungen dargestellt werden:

```
/* Ausgabe von bcd-Zahlen */
/* Das Programm muß als C++ Programm compiliert werden */

#include <bcd.h>
#include <iostream.h>
#include <stdio.h>

void main(void) {
   bcd a = 12.1;
   double x = real(a); // Konvertierung für printf().
   printf("\na = %g", x);
   printf("\na = %Lg", real(a));
   printf("\na = %g", (double)real(a));
   cout < "\na = " < a; // Die beste Methode
   }
```

Da *printf* keine Überprüfung der Argumente durchführt, muß zusätzlich zur Formatierungsanweisung das *L* angegeben werden, wenn der **long double**-Wert *real(a)* übergeben wird.

Anzahl der Dezimalstellen

Sie können angeben, wieviele Stellen hinter dem Dezimalpunkt bei der Konvertierung einer binären Zahl in einen *bcd*-Typ übernommen werden sollen. Die Anzahl der Stellen ist ein optionales zweites Argument des *bcd*-Konstruktor. Betrachten Sie dazu folgendes Beispiel: mit

```
bcd a = bcd(1000.00/7, 2)
```

kann der Betrag DM 1000.00/7 in eine *bcd*-Variable konvertiert werden, wobei auf den nächsten Pfennig gerundet wird. Die Ziffer 2 legt fest, daß zwei Dezimalstellen übernommen werden. Daraus ergibt sich:

```
1000.00/7              =     142.85714
bcd(1000.00/7, 2)      =     142.860
bcd(1000.00/7, 1)      =     142.900
bcd(1000.00/7, 0)      =     143.000
bcd(1000.00/7, -1)     =     140.000
bcd(1000.00/7, -2)     =     100.000
```

Die Rundungsmethode wird durch die IEEE genormt.

Die Rundungsmethode entspricht dem Verfahren der kaufmännischen Rechnung: Es wird auf die nächste ganze Zahl – im Zweifelsfall auf eine gerade Zahl – gerundet:

```
bcd(12.335, 2)         =     12.34
bcd(12.345, 2)         =     12.34
bcd(12.355, 2)         =     12.36
```

K A P I T E L 3

Bildschirmgrafik

Borland C++ enthält eine komplette Bibliothek mit Grafikfunktionen, mit deren Hilfe Sie Tabellen und Diagramme auf dem Bildschirm erzeugen können. Diese Grafikfunktionen sind nur für reine 16-Bit DOS-Anwendungen verfügbar. In diesem Kapitel finden Sie eine kurze Erläuterung der Video-Modi und Fenster. Außerdem wird die Programmierung im Grafikmodus erläutert.

Erläuterungen zu den Video-Modi

Ein PC ist normalerweise mit einem der folgenden Bildschirmadapter ausgestattet:

- **Monochrome Display Adapter**, MDA (nur für die Anzeige von Text);
- **Color Graphics Adapter**, CGA;
- **Enhanced Graphics Adapter**, EGA;
- **Video Graphics Array Adapter**, VGA;
- **Hercules Monochrome Graphics Adapter**.

Jeder dieser Adapter kann in zahlreichen verschiedenen Modi betrieben werden. Ein Modus bestimmt, ob auf dem Bildschirm Text in 80 oder in 40 Spalten dargestellt wird (nur in den Textmodi), welche Grafikauflösung verwendet wird (nur in den Grafikmodi) und ob Farb- oder Schwarz-/Weiß-Darstellung benutzt wird.

Der Betriebsmodus für den Bildschirm wird definiert, wenn Ihr Programm eine der drei folgenden Modusdefinitionen aufruft: *textmode*, *initgraph* oder *setgraphmode*.

- Im *Textmodus* ist der PC-Bildschirm in Zellen eingeteilt (80 oder 40 Spalten breit und 25, 43 oder 50 Zeilen hoch). Jede Zelle besteht aus einem Attribut und einem Zeichen. Das Zeichen wird als ASCII-Zeichen angezeigt; das Attribut legt fest, wie das Zeichen dargestellt wird (Farbe, Intensität etc.). Borland C++ stellt eine Reihe von Funktionen zur Verfü-

gung, mit deren Hilfe der Textbildschirm bearbeitet und Text direkt auf dem Bildschirm geschrieben werden kann. Außerdem gibt es Funktionen zur Steuerung der Zellenattribute.

- Im *Grafikmodus* ist der Bildschirm in sogenannte *Pixel* unterteilt. Jedes Pixel wird als Punkt auf dem Bildschirm dargestellt. Die Anzahl der Pixel (die sogenannte *Auflösung*) hängt vom Typ des installierten Bildschirmadapters und dem für ihn eingestellten Modus ab. Mit Hilfe von Funktionen aus der Borland C++ Grafikbibliothek können Grafiken auf den Bildschirm erzeugt werden, wie z.B. durch das Zeichnen von Linien, das Füllen von geschlossenen Flächen mit Mustern und der Bestimmung der Farbe für jedes einzelne Pixel.

Im Textmodus ist die obere linke Ecke des Bildschirms als Position (1,1) festgelegt, mit X-Koordinaten, die von links nach rechts, und mit Y-Koordinaten, die vom oberen zum unteren Bildschirmrand zunehmen. Im Grafikmodus ist die obere linke Ecke als Position (0,0) definiert, deren X- und Y-Koordinatenwerte sich wie im Textmodus verhalten.

Text- und Grafikfenster

Borland C++ stellt Funktionen zur Verfügung, mit deren Hilfe Fenster im Textmodus erstellt und verwalteter werden können (im Grafikmodus gibt es analog dazu Grafikfenster). Wenn Sie mit Text- und Grafikfenstern nicht vertraut sind, sollten Sie die folgenden Abschnitte lesen. Eine Erläuterung der Funktionen für Text- und Grafikfenster finden Sie im Abschnitt »Programmieren im Grafikmodus«.

Ein *Fenster* ist – im Textmodus – ein definierter, rechteckiger Ausschnitt auf Ihrem Bildschirm. Bildschirmausgaben Ihres Programms erfolgen nur in das aktive Fenster. Der Rest des Bildschirms (außerhalb des Fensters) bleibt von der Textausgabe unberührt.

Das voreingestellte Fenster umfaßt den gesamten Bildschirm. In Ihrem Programm können Sie diese Fenstergröße jedoch ändern und ein Textfenster definieren, das kleiner als der gesamte Bildschirm ist. Dazu rufen Sie die Funktion *window* auf, die die Position des Fensters mit Hilfe der Bildschirmkoordinaten festlegt.

Im Grafikmodus können Sie ebenfalls einen rechteckigen Ausschnitt auf Ihrem Bildschirm definieren, der als Grafikfenster (*Viewport*) bezeichnet wird. Wenn Ihr Grafikprogramm Zeichnungen usw. ausgibt, fungiert das Grafikfenster als virtueller Bildschirm. Der Rest des Bildschirms (außerhalb des Grafikfensters) bleibt davon unberührt. Ein Grafikfenster wird über die Bildschirmkoordinaten definiert. Dazu rufen Sie die Funktion *setviewport* auf.

Programmieren im Grafikmodus

In diesem Abschnitt finden Sie einen Überblick über die Funktionen für den Grafikmodus. Weitere Informationen zu diesen Funktionen finden Sie in Kapitel 4.

Borland C++ verfügt über eine separate Bibliothek mit über 70 Grafikfunktionen, die von »high level«-Funktionen (wie *setviewport* und *bar3d* und *drawpoly*) bis zu bit-orientierten Funktionen (wie *getimage* und *putimage*) reichen. Die Grafikbibliothek unterstützt eine Vielzahl von Füll- und Linienmustern und stellt mehrere Schriften zur Verfügung, die in der Größe geändert, justiert und vertikal bzw. horizontal ausgerichtet werden können.

Sämtliche Funktionen des Grafikpakets sind in einer Bibliothek namens GRAPHICS.LIB zusammengefaßt und in der Datei graphics.h als Prototypen deklariert. Zusätzlich zu diesen beiden Dateien beinhaltet das Grafikpaket Grafiktreiber (*.BGI-Dateien) und Vektor-Schriften (*.CHR-Dateien), die wir Ihnen in den folgenden Abschnitten näher vorstellen.

Wenn Sie mit dem Kommandozeilencompiler BCC.EXE arbeiten, müssen Sie GRAPHICS.LIB in der Kommandozeile aufführen. Wenn beispielsweise Ihr Programm MYPROG.C mit Grafiken arbeitet, muß die Kommandozeile folgendermaßen aussehen:

```
BCC MYPROG GRAPHICS.LIB
```

Da Grafikfunktionen **far**-Zeiger benutzen, werden Grafiken mit dem Speichermodell Tiny nicht unterstützt.

Es gibt nur eine einzige Grafikbibliothek, also keine verschiedenen Versionen für die einzelnen Speichermodelle (wie es bei den speichermodell-spezifischen Standardbibliotheken CS.LIB, CC.LIB, CM.LIB. etc. der Fall ist). Alle Funktionen in GRAPHICS.LIB sind **far**-Funktionen. Diejenigen Funktionen, die Zeiger übernehmen, übernehmen **far**-zeiger. Für die korrekte Ausführung dieser Funktionen ist es wichtig, daß `#include <graphics.h>` in jedem Modul angegeben wird, das Grafiken verwendet.

Die Funktionen der Grafikbibliothek

Die Grafikfunktionen von Borland C++ lassen sich in sieben Kategorien einteilen:

- Steuerung des Grafiksystems
- Zeichnen und Füllen
- Manipulation von Bildschirm-Grafikfenstern
- Textausgabe
- Farbsteuerung
- Fehlerbehandlung
- Statusabfragen

Steuerung des Grafiksystems

Im folgenden finden Sie eine Zusammenstellung der Funktionen zur Steuerung des Grafiksystems:

closegraph	Schließt das Grafiksystem.
detectgraph	Überprüft die Hardware und stellt fest, welcher Grafiktreiber zu verwenden ist, und schlägt einen Modus vor.
graphdefaults	Setzt alle Grafiksystemvariablen auf die voreingestellten Werte.
_graphfreemem	Gibt den vom Grafiksystem belegten dynamischen Speicher wieder frei; Möglichkeit zur Definition einer eigenen Routine.
_graphgetmem	Belegt dynamischen Speicher für das Grafiksystem; Möglichkeit zur Definition einer eigenen Routine.
getgraphmode	Liefert den aktuellen Grafikmodus zurück.
getmoderange	Liefert den »höchsten« und den »niedrigsten« Modus für den angegebenen Treiber.
initgraph	Initialisiert das Grafiksystem und schaltet die Hardware in den Grafikmodus um.
installuserdriver	Installiert einen zusätzlich erworbenen Treiber in der BGI-Gerätetreibertabelle.
installuserfont	Lädt eine zusätzlich erworbene Vektor-Schriftdatei in die BGI-Gerätetreibertabelle.
registerbgidriver	Trägt eine gelinkte oder vom Benutzer geladene Treiberdatei zum Zeitpunkt des Linkens ein.
restorecrtmode	Stellt den ursprünglichen Bildschirmmodus (vor dem Aufruf von *initgraph*) wieder her.
setgraphbufsize	Legt die Größe des internen Grafikpuffers fest.
setgraphmode	Wählt den angegebenen Grafikmodus aus, löscht den Bildschirm und stellt alle Voreinstellungen wieder her.

Das Borland C++ Grafikpaket unterstützt die folgenden Grafik-Adapter (und voll kompatible Adapter):

- Color/Graphics Adapter (CGA)
- Multi-Color Graphics Array (MCGA)
- Enhanced Graphics Adapter (EGA)
- Video Graphics Array (VGA)
- Herkules Graphics Adapter
- AT&T 400-line Graphics Adapter
- 3270 PC Graphics Adapter
- IBM 8514 Graphics Adapter

Zum Start des Grafiksystems rufen Sie zuerst die Funktion *initgraph* auf, die den Grafiktreiber lädt und das System in den Grafikmodus setzt.

Sie können *Initgraph* anweisen, einen bestimmten Grafiktreiber zu verwenden oder den zur Laufzeit installierten Video-Adapter festzustellen

und dann den entsprechenden Treiber zu aktivieren. Wenn *initgraph* den Video-Adapter feststellen soll, ruft sie die Funktion *detectgraph* auf, um damit einen Grafiktreiber und Modus auszuwählen. Wenn *initgraph* angewiesen wird, einen bestimmten Grafiktreiber zu verwenden, müssen Sie sicherstellen, daß die entsprechende Hardware vorhanden ist. Wenn *initgraph* auf nicht verfügbare Hardware zuzugreifen versucht, sind die Folgen nicht absehbar.

Sobald ein Grafiktreiber geladen ist, können Sie mit *getdrivername* den Namen des Treibers ausfindig machen. Mit *getmaxmode* erfahren Sie, wieviele Modi der Treiber unterstützt. *getgraphmode* informiert Sie darüber, welcher Grafikmodus gerade aktiv ist. Ausgehend von einer Modusnummer, können Sie den Namen des Modus mit Hilfe von *getmodename* erfahren. *setgraphmode* wechselt den Grafikmodus, und *restorecrtmode* setzt den Video-Modus in seinen ursprünglichen Zustand (vor der Initialisierung des Grafiksystems) zurück. *restorecrtmode* stellt den Bildschirm zurück in den Textmodus, schließt aber nicht das Grafiksystem (Schriften und Treiber bleiben weiterhin im Speicher).

graphdefaults setzt für die Grafikeinstellungen (Grafikfenstergröße, Zeichenfarbe, Füllfarbe und Muster) wieder die Standardwerte ein.

Mit *installuserdriver* und *installuserfont* können Sie neue Treiber und Schriften in Ihre BGI-Tabelle einfügen.

Am Ende Ihrer Grafikbearbeitung rufen Sie *closegraph* auf, um das Grafiksystem zu schließen. *closegraph* entfernt den Treiber aus dem Speicher und stellt den ursprünglichen Video-Modus (mit *restorecrtmode*) wieder her.

Eine detaillierte Erläuterung

Die vorausgehende Diskussion war ein Überblick über die Funktionsweise von *initgraph*. Im folgenden wollen wir *initgraph*, *_graphgetmem* und *_graphfreemem* näher betrachten.

Zum Laden eines Grafiktreibers reserviert *initgraph* zuerst Speicherplatz. Danach wird die entsprechende .BGI-Datei von Diskette/Platte eingelesen. Eine Alternative zu diesem dynamischen Ladevorgang ist das Linken eines (oder mehrerer) Grafiktreiber(s) direkt in die ausführbare Programmdatei. Dazu müssen Sie zuerst die .BGI-Datei in eine .OBJ-Datei konvertieren (mit dem Zusatzprogramm BGIOBJ – vgl. dazu die Online-Dokumentation UTIL.DOC auf den Borland C++ Originaldisketten). Danach fügen Sie Aufrufe von *registerbgidriver* in den Quelltext ein (vor dem Aufruf von *initgraph*), damit der/die Grafiktreiber registriert werden.

Wenn Sie Ihr Programm aufbauen, müssen Sie die .OBJ-Dateien für die registrierten Treiber linken.

Nachdem (mit *detectgraph*) festgestellt wurde, welcher Grafiktreiber verwendet wird, überprüft *initgraph*, ob der gewünschte Treiber vorhanden

ist. Wenn das der Fall ist, verwendet *initgraph* den registrierten Treiber direkt aus dem Speicher. Andernfalls reserviert *initgraph* Speicherplatz für den Treiber und lädt die .BGI-Datei von Platte/Diskette.

Hinweis: *registerbgidriver* sollte nur von erfahrenen Programmierern verwendet werden. Diese Funktion ist im Kapitel 4 detailliert erläutert.

Während der Laufzeit kann das Grafiksystem gezwungen sein, Speicherplatz für Treiber, Schriften und interne Puffer zu reservieren. Wenn dies notwendig ist, wird *_graphgetmem* zur Speicherplatzbelegung und *_graphfreemem* zur Speicherplatzfreigabe aufgerufen. Per Voreinstellung rufen diese Routinen *malloc* und *free* auf.

Wenn Sie eigene Funktionen dieses Namens verwenden, erhalten Sie möglicherweise eine Warnung über doppelte Symbole, die Sie ignorieren können.

Dieses voreingestellte Verhalten der Funktionen können Sie durch selbstdefinierte *_graphgetmem* und _graphfreemem-Funktionen überschreiben. Wenn Sie die Funktionen in Ihrem Sinne ändern, können Sie die Speicherplatzbelegung selbst steuern. Sie müssen Ihren Routinen jedoch die bestehenden Namen geben. Dadurch werden die Standardfunktionen mit denselben Namen in den C-Standardbibliotheken überschrieben.

Zeichnen und Füllen

Im folgenden finden Sie eine Übersicht über die Zeichen- und Füllfunktionen.

Zeichnen:

arc	Zeichnet einen Kreisbogen.
circle	Zeichnet einen Kreis.
drawpoly	Zeichnet den Umriß eines Vielecks.
ellipse	Zeichnet einen Ellipsenbogen.
getarccoords	Liefert die Koordinaten des letzten Aufrufs von *arc* oder *ellipse* zurück.
getaspectratio	Liefert den Korrekturfaktor für das physikalische Höhen-/Seitenverhältnis des Grafikmodus zurück.
getlinesetting	Liefert die aktuelle Linienart, das aktuelle Linienmuster und die aktuelle Linienstärke zurück.
line	Zeichnet eine Linie von (*x0,y0*) zu (*x1,y1*).
linerel	Zeichnet eine Linie zu einem Punkt mit einer relativen Entfernung von der aktuellen Position (CP).
lineto	Zeichnet eine Linie von der aktuellen Position (CP) zu (*x,y*).
moveto	Verschiebt die aktuelle Position (CP) zu (*x,y*).
moverel	Verschiebt die aktuelle Position (CP) um einen relativen Betrag.
rectangle	Zeichnet ein Rechteck.
setaspectratio	Ändert den voreingestellten Korrekturfaktor für das Höhen-/Seitenverhältnis.
setlinestyle	Bestimmt die aktuelle Linienbreite und -art.

Füllen:	*bar*	Zeichnet und füllt einen Balken.
	bar3d	Zeichnet und füllt einen dreidimensionalen Balken.
	fillellipse	Zeichnet und füllt eine Ellipse.
	fillpoly	Zeichnet und füllt ein Vieleck.
	floodfill	Füllt eine umschlossene Fläche.
	getfillpattern	Liefert das benutzerdefinierte Füllmuster zurück.
	getfillsettings	Liefert Informationen über das aktuelle Füllmuster und die aktuelle Füllfarbe.
	pieslice	Zeichnet und füllt ein »Tortenstück«.
	sector	Zeichnet und füllt ein elliptisches »Tortenstück«.
	setfillpattern	Wählt ein benutzerdefiniertes Füllmuster aus.
	setfillstyle	Legt Füllmuster und Füllfarbe fest.

Mit Hilfe der Zeichen- und Füllfunktionen können Sie farbige Linien, Kreisbögen, Kreise, Ellipsen, Rechtecke, »Tortenstücke«, zwei- und dreidimensionale Balken, Vielecke sowie regelmäßige und unregelmäßige Figuren durch Kombination der vorhandenen Funktionen zeichnen. Sie können jede umschlossene Fläche (oder jeden Bereich, der eine umschlossene Fläche umgibt) mit einem der 11 vordefinierten oder einem selbstdefinierten Muster ausfüllen. Außerdem können die Linienstärke und -art und die aktuelle Position (PC) festgelegt werden.

Linien und nicht ausgefüllte Formen werden mit den Funktionen *arc*, *circle*, *drawpoly*, *ellipse*, *line*, *linerel*, *lineto* und *rectangle* gezeichnet. Diese Formen können Sie mit *floodfill* ausfüllen. Dies erreichen Sie aber auch in einem Arbeitsschritt, indem Sie die Funktionen *bar, bar3d, fillellipse, fillpoly, pieslice* und *sector* benutzen. Mit *setlinestyle* bestimmen Sie, wie dick oder dünn die gezeichneten Linien (und die Einfassungen von ausgefüllten Flächen) sein sollen. Außerdem können Sie festlegen, ob die Linien durchgezogen, gepunktet etc. sein oder ein selbstdefiniertes Muster annehmen sollen. Mit *setfillstyle* wählen Sie ein vordefiniertes Füllmuster aus, und mit *setfillpattern* definieren Sie Ihr eigenes Muster. Die aktuelle Position können Sie mit *moveto* an eine angegebene Stelle oder mit *moverel* um einen bestimmten relativen Abstand verschieben.

Wenn Sie *getlinesettings* aufrufen, erfahren Sie die aktuelle Linienart und -stärke. Informationen über das aktuelle Füllmuster und die aktuelle Füllfarbe erhalten Sie mit *getfillsettings*. Mit *getfillpattern* werden Sie über das selbstdefinierte Füllmuster informiert.

Den Korrekturfaktor für das Höhen-/Seitenverhältnis (das ist der Faktor, den das Grafiksystem benötigt, um das runde Aussehen der Kreise sicherzustellen) erhalten Sie mit *getaspectratio* und die Koordinaten des zuletzt gezeichneten Kreisbogens oder der zuletzt gezeichneten Ellipse mit *getarccoords*. Wenn Kreise nicht exakt rund werden, können Sie sie mit *setaspectratio* korrigieren.

Manipulation von Bildschirm und Grafikfenster

Im folgenden finden Sie eine Übersicht über die Funktionen zur Manipulation von Bildschirm, Grafikfenster, Bildern und Pixeln.

Bildschirmmanipulation:	*cleardevice*	Löscht den Bildschirm (die aktive Seite).
	setacttivpage	Bestimmt die aktive Seite für die Ausgabe von Grafiken.
	setvisualpage	Gibt die Nummer der Seite an, die angezeigt werden soll.
Grafikfenstermanipulation:	*clearviewport*	Löscht aktuelle Grafikfenster.
	getviewsettings	Liefert Informationen über das aktuelle Grafikfenster.
	setviewport	estimmt das aktuelle Grafikfenster für die Grafikausgabe.
Bildmanipulation:	*getimage*	Sichert ein Bit-Bild in einen angegebenen Bereich im Speicher.
	imagesize	Liefert die Anzahl der Bytes, die für das Speichern eines rechteckigen Bildschirmausschnitts erforderlich sind.
	putimage	Bringt ein vorher gespeichertes Bit-Bild auf den Bildschirm.
Pixelmanipulation:	*getpixel*	Liefert die Farbe eines Pixel an der Position (*x*,*y*).
	putpixel	Zeichnet ein einzelnes Pixel an der Position (*x*,*y*).

Neben den Funktionen zum Zeichnen und Füllen beinhaltet die Grafikbibliothek Funktionen zur Manipulation von Bildschirm, Grafikfenstern, Bildern und Pixeln. Sie können mit dem Aufruf von *cleardevice* den gesamten Bildschirm in einem Arbeitsschritt löschen. Dabei wird die aktuelle Position in den Ursprung (Home-Position) des Grafikfensters gebracht; alle anderen Grafikeinstellungen bleiben unverändert aktiv (Linien-, Füllmuster und Textart; die Palette, die Einstellungen des Grafikfensters etc.).

In Abhängigkeit von Ihrem Grafik-Adapter verfügt Ihr System über Bildschirmpuffer, die eine bis vier Seiten umfassen. Dabei handelt es sich um Bereiche im Speicher, in denen einzelne Ganzseitenbilder punktweise abgespeichert sind. Sie können die aktive Bildschirmseite (die Seite, auf die die Grafikfunktionen ihre Ausgaben bringen sollen) mit *setactivepage* bestimmen. Die Seite, die auf dem Bildschirm angezeigt werden soll, legen Sie mit *setvisualpage* fest.

Wenn sich der Bildschirm im Grafikmodus befindet, können Sie mit *setviewport* ein Grafikfenster definieren (einen rechteckigen »virtuellen Bildschirm«). Die Position des Grafikfensters wird durch die Angabe von absoluten Bildschirmkoordinaten festgelegt. Sie können außerdem festlegen, ob die Clipping-Funktionen (das Abschneiden am Bildschirmrand) ein- oder ausgeschaltet sein soll. Mit *clearviewport* löschen Sie das Grafikfenster. Wenn Sie die aktuellen absoluten Bildschirmkoordinaten des Grafikfensters und den Clipping-Status wissen wollen, rufen Sie *getviewsettings* auf.

Mit *getimage* können Sie einen Teil eines Bildschirmbildes speichern, mit *imagesize* die Anzahl der Bytes ausrechnen lassen, die für das Abspei-

chern des Bildes im Speicher erforderlich sind, und das Bild dann wieder mit *putimage* (an eine beliebige Stelle) auf den Bildschirm zurückbringen.

Die Koordinaten für alle Ausgabefunktionen (Zeichen, Füllen, Text etc.) sind relativ zum Grafikfenster.

Mit den Funktionen *getpixel* und *putpixel* können Sie die Farbe einzelner Pixel bestimmen. *getpixel* liefert die Farbe eines angegebenen Pixel, und *putpixel* zeichnet ein bestimmtes Pixel in einer angegebenen Farbe.

Textausgabe und Grafikmodus

Im folgenden finden Sie eine Übersicht über die Funktionen zur Textausgabe im Grafikmodus.

gettextsettings	Liefert die aktuelle Schrift, Orientierung, Größe und Ausrichtung.
outtext	Gibt einen String an der aktuellen Position auf dem Bildschirm aus.
outtextxy	Gibt einen String an der angegebenen Position auf dem Bildschirm aus.
registerbgifont	Registriert eine gelinkte oder vom Benutzer geladene Schrift.
settextjustify	Bestimmt Modi für die Textausrichtung, die von den Funktionen *outtext* und *outtextxy* verwendet werden.
settextstyle	Bestimmt die aktuelle Schrift, die aktuelle Art und den Faktor zur Vergrößerung der Zeichen.
setusercharsize	Definiert das Verhältnis von Höhe und Breite bei Vektorschriften.
textheight	Liefert die Höhe eines Strings in Pixel.
textwidth	Liefert die Breite eines Strings in Pixel.

Die Grafikbibliothek beinhaltet eine 8x8-Bitmap-Schrift und mehrere Vektorschriften für die Textausgabe im Grafikmodus.

- In einer Bitmap-Schrift wird jedes Zeichen durch eine Pixel-Matrix definiert.
- In einer Vektorschrift wird jedes Zeichen durch Vektoren definiert, die dem Grafiksystem mitteilen, wie dieses Zeichen gezeichnet werden muß.

Der Vorteil von Vektorschriften wird deutlich, wenn Sie große Zeichen zeichnen. Da eine Vektorschrift über Vektoren definiert ist, bleibt eine qualitativ hochwertige Auflösung erhalten, wenn das Zeichen vergrößert wird. Wenn Sie aber eine Bitmap-Schrift vergrößern, wird die Matrix mit einem Änderungsfaktor multipliziert; wird der Änderungsfaktor größer, die Zeichenauflösung gröber. Bei kleinen Zeichen ist eine Bitmap-Schrift ausreichend; für größere Textzeichen sollten Sie aber eine Vektorschrift wählen.

Im Grafikmodus wird Text mit Hilfe von *outtext* oder *outtextxy* ausgegeben. Die Textausrichtung wird (in Bezug auf die aktuelle Position) mit *settextjustify* gesteuert. Mit *settextstyle* legen Sie die Schrift, die Orientierung (horizontal oder vertikal) und die Größe (Skalierung) fest. Die aktuellen Texteinstellungen erfahren Sie durch einen Aufruf von *gettextsettings*. Die Funktion liefert die aktuelle Schriftart, Ausrichtung, Größe und Orientierung in einer Struktur zurück. *setusercharsize* ermöglicht die Änderung von Breite und Höhe bei Vektorschriftzeichen.

Wenn die Clipping-Funktion eingeschaltet ist, wird der mit *outtext* oder *outtextxy* ausgegebene Text an den Rändern der Grafikfenster abgeschnitten. Ist Clipping nicht aktiv, verwerfen diese Funktionen bei einer Bitmap-Schrift den gesamten Text, wenn Teile der Textstrings über den Bildschirmrand hinaus reichen würden. Textausgaben mit einer Vektorschrift werden am Bildschirmrand abgeschnitten.

Wenn Sie die Größe eines Textstrings auf dem Bildschirm feststellen möchten, rufen Sie *textheight* und *textwidth* auf. *textheight* liefert die Höhe des Strings und *textwidth* die Breite des Strings in Pixel.

Die voreingestellte 8x8-Bitmap-Schrift ist im Grafikpaket enthalten, so daß sie zur Laufzeit immer zur Verfügung steht. Die Vektorschriften liegen in separaten .CHR-Dateien vor; sie können zur Laufzeit geladen oder aber in .OBJ-Dateien konvertiert (mit dem Zusatzprogramm BGIOBJ) und in die .EXE-Datei gelinkt werden.

Zum Laden einer Schriftdatei reserviert *settextstyle* zuerst Speicher für die Schrift. Danach wird die entsprechende .CHR-Datei von Platte/Diskette geladen. Eine Alternative zum Laden zur Laufzeit ist das Linken von einer (oder mehreren) Schriftdatei(en) direkt in die ausführbare Datei. Sie müssen zuerst die .CHR-Datei in eine .OBJ-Datei konvertieren (mit dem Zusatzprogramm BGIOBJ – vgl. Sie dazu die Online-Dokumentation auf den Borland C++ Originaldisketten). Danach fügen Sie Aufrufe von *registerbgifont* in den Quelltext ein (vor dem Aufruf von *settextstyle*), um die Schriften einzutragen. Wenn Sie Ihr Programm aufbauen, müssen Sie die .OBJ-Dateien für die eingetragenen Vektorschriften linken.

Hinweis

registerbgifont sollte nur von erfahrenen Programmierern verwendet werden. Diese Funktion finden Sie in UTIL.DOC auf Ihren Originaldisketten detailliert erläutert.

Farbsteuerung

Es folgt eine Übersicht über die Funktionen zur Farbsteuerung.

Informationen über die Farbgebung:

getbkcolor	Liefert die aktuelle Hintergrundfarbe.
getcolor	Liefert die aktuelle Zeichenfarbe.
getdefaultpalette	Liefert die Struktur der Palettendefinition.
getmaxcolor	Liefert den maximalen Farbwert, der im aktuellen Grafikmodus zur Verfügung steht.

	getpalette	Liefert die aktuelle Palette und ihre Größe.
	getpalettesize	Liefert die Größe der Palettentabelle.
Bestimmung einer oder mehrerer Farben:	*setallpalette*	Ändert alle Farben der Palette wie angegeben.
	setbkcolor	Legt die aktuelle Hintergrundfarbe fest.
	setcolor	Bestimmt die aktuelle Farbe für das Zeichnen.
	setpalette	Ändert eine Palettenfarbe, wie in den Argumenten definiert.

Bevor wir diese Funktionen zusammenfassend erläutern, wollen wir einige grundlegende Informationen über die Erzeugung von Farben auf Ihrem Grafikbildschirm anfügen.

Pixel und Paletten

Der Grafikbildschirm besteht aus einem Bereich mit Pixeln; jedes Pixel ist ein (farbiger) Punkt auf dem Bildschirm. Der Wert eines Pixel legt aber nicht direkt die genaue Farbe fest; er ist vielmehr ein Eintrag in einer Farbtabelle, die *Palette* genannt wird. Der Eintrag in der Palette, der einem gegebenen Pixelwert entspricht, enthält die exakte Farbinformation für das betreffende Pixel.

Aufgrund dieses indirekten Vorgehens gibt es folgende Implikationen: Obwohl die Hardware möglicherweise viele Farben anzeigen kann, ist zu einem gegebenen Zeitpunkt nur eine bestimmte Menge dieser Farben darstellbar. Die Anzahl der Farben, die zu einem Zeitpunkt dargestellt werden können, entspricht der Anzahl der Einträge in der Palette (der Palette*größe*). Mit einem EGA-Adapter können beispielsweise 64 Farben dargestellt werden, aber nur 16 davon zur gleichen Zeit. Die Palettengröße für einen EGA-Adapter beträgt somit 16.

Die *Größe* der Palette bestimmt die Rangfolge der Werte, die ein Pixel annehmen kann, und zwar von 0 bis (*Größe* - 1). *getmaxcolor* liefert den höchsten zulässigen Pixelwert (*Größe* - 1) für den aktuellen Treiber und Modus.

Bei der Erläuterung der Borland C++ Grafikfunktionen wird häufig der Ausdruck *Farbe* verwendet, wie z.B. in aktuelle Zeichenfarbe, Füllfarbe und Pixelfarbe. Tatsächlich ist diese Farbe der Wert des Pixel: ein Eintrag in der Palette. Nur die Palette legt die tatsächliche Farbe auf dem Bildschirm fest. Durch Manipulation der Palette kann die auf dem Bildschirm angezeigte Farbe modifiziert werden, ohne die Pixelwerte (Zeichenfarbe, Füllfarbe etc.) zu ändern.

Hintergrund- und Zeichenfarbe

Die *Hintergrundfarbe* entspricht immer dem Pixelwert 0. Wenn in einem Bereich die Hintergrundfarbe gelöscht wird, werden die Pixel dieses Bereichs einfach auf 0 gesetzt.

Die *Zeichenfarbe* ist der Wert, auf den die Pixel beim Zeichnen von Linien gesetzt werden. Eine Farbe für das Zeichnen wird mit der Funktion

setcolor(n) ausgewählt, wobei *n* ein zulässiger Pixelwert in der aktuellen Palette ist.

Farbsteuerung bei CGA-Adaptern

Wegen der Unterschiede bei der Hardware ist auch die Farbsteuerung bei CGA- und EGA-Adaptern nicht ganz identisch; wir werden deshalb in getrennten Erläuterungen darauf eingehen. Die Farbsteuerung bei einem AT&T-Treiber und die niedrigere Auflösung bei MCGA-Treibern entsprechen den CGA-Adaptern.

Bei CGA-Adaptern können Sie wählen, ob Ihre Grafiken mit niedrigerer Auflösung (320x200) unter Verwendung von vier Farben oder mit hoher Auflösung (640x200) unter Verwendung von zwei Farben angezeigt werden sollen.

Niedrige Auflösung bei CGA-Adaptern

Bei niedriger Auflösung können Sie eine von vier vordefinierten Paletten mit je vier Farben wählen. In all diesen Paletten ist nur der erste Eintrag veränderbar, die Einträge 1, 2 und 3 sind fest. Der erste Eintrag in der Palette (Farbe 0) bezieht sich auf die Hintergrundfarbe. Für den Hintergrund kann eine der 16 verfügbaren Farben gewählt werden (vgl. die nachfolgende Tabelle mit den CGA-Hintergrundfarben).

Sie wählen die gewünschte Palette durch Anwahl des entsprechenden Modus (CGAC0, CGAC1, CGAC2, CGAC3) aus; diese Modi setzen die Farbpaletten 0 bis 3 ein, wie aus der folgenden Tabelle ersichtlich wird. Die CGA-Zeichenfarben und die entsprechenden Konstanten sind in graphics.h definiert.

Nr. der Palette	Die der Farbnummer (Pixelwert) zugewiesene Konstante		
	1	2	3
0	CGA_LIGHTGREEN	CGA_LIGHTRED	CGA_YELLOW
1	CGA_LIGHTCYAN	CGA_LIGHTMAGENTA	CGA_WHITE
2	CGA_GREEN	CGA_RED	CGA_BROWN
3	CGA_CYAN	CGA_MAGENTA	CGA_LIGHTGRAY

Wenn Sie eine dieser Farben als Zeichenfarbe zuweisen wollen, rufen Sie *setcolor* auf und vergeben als Argument entweder die Farbnummer oder den entsprechenden Konstantennamen. Wenn Sie z.B. Palette 3 verwenden und Cyan als Hintergrundfarbe einsetzen wollen, dann geben Sie folgendes ein:

```
setcolor (1);
```

oder

```
setcolor (CGA_CYAN);
```

In der folgenden Tabelle sind die in graphics.h definierten CGA-Hintergrundfarben zusammengestellt:

Die CGA-Vordergrundfarben entsprechen den aufgeführten Hintergrundfarben.

Numerischer Wert	Symbolischer Name	Farbe
0	BLACK	Schwarz
1	BLUE	Blau
2	GREEN	Grün
3	CYAN	Türkis
4	RED	Rot
5	MAGENTA	Fuchsinrot
6	BROWN	Braun
7	LIGHTGRAY	Hellgrau
8	DARKGRAY	Dunkelgrau
9	LIGHTBLUE	Hellblau
10	LIGHTGREEN	Hellgrün
11	LIGHTCYAN	helles Türkis
12	LIGHTRED	Hellrot
13	LIGHTMAGENTA	helles Fuchsinrot
14	YELLOW	Gelb
15	WHITE	Weiß

Wenn Sie eine dieser Farben für den Hintergrund einsetzen möchten, rufen Sie *setbkcolor(Farbe)* auf, wobei *Farbe* einer der oben aufgelisteten Tabelleneinträge ist. Bei CGA-Adaptern entspricht die Farbe nicht einem Pixelwert (Paletteneintrag); die Farbe legt direkt die *tatsächliche* Farbe fest, die als erster Eintrag in die Palette eingebracht werden soll.

Hohe Auflösung bei CGA-Adaptern

Bei hoher Auflösung stehen zwei Farben zur Verfügung: schwarzer Hintergrund und farbiger Vordergrund. Die Pixel können den Wert 0 oder 1 annehmen. Aufgrund einer (seltsamen) Unzulänglichkeit bei CGA-Adaptern ist die Vordergrundfarbe diejenige, die die Hardware als Hintergrundfarbe erkennt. Die Einstellung erfolgt mit der Funktion *setbkcolor*.

Die für den Vordergrund zur Verfügung stehenden Farben entnehmen Sie der obigen Tabelle. Der CGA-Adapter verwendet die angegebene Farbe für die Pixel, deren Wert auf 1 gesetzt ist.

Folgende Modi arbeiten auf diese Weise: CGAHI, MCGAMED, MCGAHI, ATT400MED und ATT400HI.

CGA-Palettenroutinen

Da die CGA-Palette vordefiniert ist, sollten Sie bei Verwendung eines CGA-Adapters nicht die Funktion *setallpalette* verwenden. Gleiches gilt für den Einsatz von *setpalette(Eintrag, tatsächliche_Farbe)*, ausgenommen für *Eintrag = 0*. (Dabei handelt es sich um eine Alternative für das Setzen der CGA-Hintergrundfarbe auf *tatsächliche_Farbe*.)

Farbsteuerung bei EGA- und VGA-Adaptern

Für EGA-Adapter enthält die Farbpalette 16 von insgesamt 64 möglichen Farben. Jeder Eintrag kann vom Benutzer gesetzt werden. Mit *getpalette* können Sie sich über die aktuelle Palette informieren; Sie erhalten eine Struktur, die die Palettengröße (16) und eine Liste der tatsächlichen Paletteneinträge (die in der Palette gespeicherten »Hardware-Farbnummern«) enthält. Mit *setpalette* werden die Paletteneinträge einzeln geändert, mit *setallpalette* können alle Einträge in einem Schritt modifiziert werden.

Die voreingestellte EGA-Palette entspricht den 16 CGA-Farben, die in der obigen Tabelle aufgelistet sind: Schwarz ist der Eintrag 0, blau der Eintrag 1, ... weiß der Eintrag 15. Es stehen auch in graphics.h definierte Konstanten zur Verfügung, die die entsprechenden Hardwarenummern der Farben enthalten, wie z.B. EGA_BLACK, EGA_WHITE, etc. Diese Werte erhalten Sie mit *getpalette*.

Die Routine *setbkcolor(Farbe)* arbeitet bei EGA-Adaptern anders als bei CGA-Adaptern. Im Falle von EGA-Adaptern kopiert *setbkcolor* den tatsächlichen im Eintrag-Nr. *Farbe* gespeicherten Farbwert in den Eintrag-Nr. 0.

In Bezug auf die Steuerung von Farben verhält sich der VGA-Treiber genauso wie der EGA-Treiber; er verfügt jedoch über eine höhere Auflösung (und kleinere Pixel).

Fehlerbehandlung im Grafikmodus

Im folgenden finden Sie eine Übersicht über die Funktionen zur Fehlerbehandlung im Grafikmodus.

grapherrormsg	Liefert einen Fehlermeldungsstring für den angegebenen Fehlercode.
graphresult	Liefert einen Fehlercode für die letzte Grafikoperation, bei der ein Problem aufgetreten ist.

Wenn bei einem Aufruf einer Funktion aus der Grafikbibliothek ein Fehler auftritt, wird intern ein Fehlercode gesetzt (wenn z.B. eine von *settextstyle* benötigte Schrift nicht gefunden werden kann). Mit *graphresult* können Sie den Fehlercode für einen bei der letzten Grafikoperation aufgetretenen Fehler in Erfahrung bringen. Ein Aufruf von *grapherrormsg(graphresult()* liefert den Fehlerstring. Die zurückgelieferten Fehlerstrings sind in der folgenden Tabelle aufgeführt.

Der Fehlerrückgabewert wird akkumuliert und nur geändert, wenn eine Grafikfunktion einen Fehler meldet. Der Fehlerrückgabecode wird nur dann auf 0 gesetzt, wenn *initgraph* erfolgreich ausgeführt oder *graphresult* aufgerufen wurde. Deshalb sollten Sie – wenn Sie wissen möchten, welche Grafikfunktion welchen Fehler zurückgeliefert hat - den Wert von *graphresult* in einer temporären Variable speichern und diese dann überprüfen.

In der folgenden Tabelle werden die Fehlercodes und die entsprechenden Strings aufgelistet.

Fehler code	*graphics_error-* Konstante	Entsprechender Fehlermeldungsstring
0	grOk	No error
-1	grNoInitGraph	(BGI) graphics not installed (use *initgraph*)
-2	grNotDetected	Graphics hardware not detected
-3	grFileNotFound	Device driver file not found
-4	grInvalidDriver	Invalid device driver file
-5	grNoLoadMem	Not enough memory to load driver
-6	grNoScanMem	Out of memory in scan file
-7	grNoFloodMem	Out of memory in flood file
-8	grFontNotFound	Font file not found
-9	grNoFontMem	Not enough memory to load font
-10	grInvalidMode	Invalid graphics mode for selected driver
-11	grError	Graphics error
-12	grIOerror	Graphics I/O error
-13	grInvalidFont	Invalid font file
-14	grInvalidFontNum	Invalid font number
-15	grInvalidDeviceNum	Invalid device number
-18	grInvalidVersion	Invalid version of file

Statusabfrage

Im folgenden finden Sie eine Übersicht über die Funktionen zur Statusabfrage im Grafikmodus:

Tabelle 3.1: Statusabfragefunktionen

Funktion	Rückgabe
getarccoords	Information über die Koordinaten des letzten Aufrufs von *arc* oder *ellipse*
getaspectratio	Korrekturfaktor für Höhen-/Seitenverhältnis im Grafikbildschirm
getbkcolor	Aktuelle Hintergrundfarbe
getcolor	Aktuelle Zeichenfarbe
getdrivername	Name des aktuellen Grafiktreibers
getfillpattern	Benutzerdefiniertes Füllmuster
getfillsettings	Information über das aktuelle Füllmuster und die aktuelle Füllfarbe
getgraphmode	Aktueller Grafikmodus
getlinesettings	Aktuelle Linienart, aktuelles Linienmuster und aktuelle Linienstärke
getmaxcolor	Höchster aktuell gültiger Pixelwert
getmaxmode	Höchste Modusnummer für den aktuellen Treiber
getmaxx	Aktuelle *x*-Auflösung
getmaxy	Aktuelle *y*-Auflösung

getmodename	Name des angegebenen Treibermodus
getmoderange	Modusbereich für den angegebenen Treiber
getpalette	Aktuelle Palette und ihre Größe
getpixel	Farbe des Pixels an Position. (*x*,*y*)
gettextsettings	Information über die aktuelle Textschrift, Orientierung, Größe und Ausrichtung
getviewsettings	Information über das aktuelle Grafikfenster
getx	X-Koordinate der aktuellen Position (CP)
gety	Y-Koordinate der aktuellen Position (CP)

Für jede Kategorie der Borland C++ Grafikfunktionen gibt es eine Statusabfragefunktion. Diese Funktionen sind bei der Erläuterung der jeweiligen Kategorie angesprochen und im folgenden kurz erläutert. Alle Statusabfragefunktionen sind nach dem Schema get*irgendetwas* benannt (ausgenommen die Fehlerbehandlung). Einige Funktionen haben keine Argumente und liefern einen einzigen Wert zurück, der die angeforderte Information enthält. Andere Funktionen benötigen einen Zeiger auf eine in graphics.h definierte Struktur und füllen diese Struktur mit den entsprechenden Informationen; sie liefern keinen Wert zurück.

Die Statusabfragefunktionen, die die Grafiksystemsteuerung betreffen, sind: *getgraphmode, getmaxmode* und *getmoderange*. Die erste Funktion liefert einen Integerwert, der den aktuellen Grafiktreiber und Modus darstellt; die zweite Funktion liefert die höchste Modusnummer für einen gegebenen Treiber und die dritte Funktion den Modusbereich, der vom gegebenen Treiber unterstützt wird. *getmaxx* und *getmaxy* liefern die höchste X- und Y-Bildschirmkoordinate für den aktuellen Grafikmodus.

Die Funktionen *getarccoords*, *getaspectratio*, *getfillpattern*, *getfillsettings* und *getlinesettings* gehören zu den Zeichen- und Füllfunktionen. *getarccoords* füllt eine Struktur mit den Koordinaten des letzten Aufrufs von *arc* und *ellipse*; *getaspectratio* informiert über den Korrekturfaktor für das Höhen-/Seitenverhältnis, den das Grafiksystem benötigt, um Kreise unverzerrt ausgeben zu können. *getfillpattern* liefert das aktuelle benutzerdefinierte Füllmuster. *getfillsettings* füllt eine Struktur mit dem Füllmuster und der aktuellen Füllfarbe aus und *getlinesettings* mit der aktuellen Linienart (durchgezogen, gestrichelt etc.), der aktuellen Linienstärke (normal oder dick) und dem aktuellen Linienmuster.

Die Statusabfragefunktionen, die zu den Bildschirm- und Grafikfensterfunktionen gehören, umfassen *getviewsettings*, *getx*, *gety* und *getpixel*. Wenn Sie ein Grafikfenster definiert haben, können Sie mit *getviewsettings* feststellen, wie die absoluten Koordinaten für das Fenster lauten und ob die Clipping-Funktion eingeschaltet ist oder nicht. Die Funktion füllt eine Struktur mit diesen Informationen aus. *getx* und *gety* liefern die X- und Y-Koordinaten der aktuellen Position (CP), und zwar in Bezug auf das Grafikfenster. *getpixel* liefert die Farbe eines angegebenen Pixels zurück.

Die Kategorie der Textausgabefunktionen verfügt über eine alle Informationen abdeckende Statusabfragefunktion: *gettextsettings*. Diese Funktion füllt eine Struktur aus, die Informationen über die aktuelle Schrift, die Orientierung der Textanzeige (für horizontal oder für vertikal), den Faktor der Zeichenvergrößerung und die Textstring-Ausrichtung (horizontal und vertikal) liefert.

Zur Funktionskategorie der Farbsteuerung gehören drei Funktionen, die den Status abfragen: *getbkcolor* liefert die aktuelle Hintergrundfarbe und *getcolor* die aktuelle Zeichenfarbe. *getpalette* füllt eine Struktur mit der Größe der aktuellen Zeichenpalette und dem Inhalt der Palette aus. *getmaxcolor* liefert den höchsten zulässigen Pixelwert für den aktuellen Grafiktreiber und Modus (Palette*größe* - 1).

Die Funktionen *getmodename* und *getdrivername* liefern den Namen eines gegebenen Treibermodus bzw. den Namen des aktuellen Grafiktreibers zurück.

K A P I T E L 4

Grafikfunktionen

In diesem Kapitel werden die Grafikfunktionen von Borland C++ in alphabetischer Reihenfolge vorgestellt. Die Grafikfunktionen sind nur für 16-Bit DOS-Anwendungen verfügbar.

arc graphics.h

Funktion Zeichnet einen Kreisbogen.

Syntax

```
void far arc(int x, int y, int stangle, int endangle, int radius);
```

Beschreibung *arc* zeichnet in der aktuellen Zeichenfarbe einen Kreisbogen mit dem Mittelpunkt (*x*,*y*), wobei *stangle* den Startpunkt und *endangle* den Endpunkt des mit dem Radius *radius* gezogenen Bogens festlegen. Wird für *stangle* der Wert 0 und für *endangle* der Wert 360 angegeben, zeichnet *arc* einen Kreis.

Die Angabe beider Winkel erfolgt in Grad, wobei entgegen dem Uhrzeigersinn gezählt wird (0 Grad = horizontal rechts vom Mittelpunkt, 90 Grad = senkrecht über dem Mittelpunkt usw.).

Der Parameter *linestyle* für die Linienart wirkt sich auf *arc*, *circle*, *ellipse*, *fillellipse*, *sector* und *pieslice* nicht aus; der Parameter *thickness* für die Liniendicke wird dagegen verwendet.

Bei Verwendung eines CGA-Systems im hochauflösenden Modus oder eines Monochrom-Grafikadapters kann es vorkommen, daß die Beispielprogramme für die Verwendung der Grafikroutinen nicht die erwarteten Ergebnisse erzeugen. In diesen Fällen sollte den Funktionen, die die Füll- oder Zeichenfarbe ändern (wie *setcolor*, *setfillstyle* und *setlinestyle*), der Wert 1 anstelle einer der symbolischen Farbkonstanten (diese sind in graphics.h definiert) übergeben werden.

Rückgabewert Keiner.

Siehe auch *circle*, *ellipse*, *fillellipse*, *getarccoords*, *getaspectratio*, *graphresult*, *pieslice*, *sector*

bar — graphics.h

Funktion Zeichnet eine zweidimensionale Leiste.

Syntax

```
#include <conio.h>
void far bar(int left, int top, int right, int bottom);
```

Beschreibung *bar* zeichnet eine gefüllte rechteckige Leiste. Verwendet wird die aktuelle Zeichenfarbe sowie das aktuelle Füllmuster. Einen Umriß zeichnet *bar* nicht – hierzu können Sie *bar3d* mit *depth* = 0 verwenden.

Die obere linke Ecke der Leiste ist durch die Koordinaten (*left*, *top*) und die untere rechte Ecke durch (*right*, *bottom*) – in Pixel – festgelegt.

Rückgabewert Keiner.

Siehe auch *bar3d*, *rectangle*, *setcolor*, *setfillstyle*, *setlinestyle*

bar3d — graphics.h

Funktion Zeichnet einen dreidimensionalen Balken.

Syntax

```
void far bar3d(int left, int top, int right, int bottom, int depth, int
topflag);
```

Beschreibung Balken, die mit *bar3d* gezeichnet werden, sind »dreidimensional« und bestehen aus einem gefüllten Rechteck, dessen obere linke Ecke durch die Koordinaten (*left*, *top*) und dessen untere rechte Ecke durch (*right*, *bottom*) – in Pixel – festgelegt ist. *bar3d* zeichnet zuerst die Umrißlinien in der aktuellen Farbe und der aktuellen Linienart, danach füllt die Funktion die umschlossene Fläche (wie *bar*) mit dem aktuellen Füllmuster aus.

Der Parameter *topflag* legt fest, ob *bar3d* einen oberen Abschluß des Balkens zeichnet (*topflag*=1) oder nicht (*topflag*=0) – im letzteren Fall lassen sich mehrere Balken übereinander stapeln. Über den Parameter *depth* läßt sich die räumliche Tiefe in Pixel bestimmen. Als Faustregel läßt sich sagen, daß die Tiefe rund 25% der Breite betragen sollte.

Dies ist ein typischer Aufruf von *bar3d*:

```
bar3d(left, top, right, bottom, (right-left) / 4, 1);
```

Rückgabewert Keiner.

Siehe auch *bar*, *rectangle*, *setcolor*, *setfillstyle*, *setlinestyle*

circle — graphics.h

Funktion Zeichnet einen Kreis mit dem angegebenen Radius um den Mittelpunkt (*x*,*y*).

Syntax

```
void far circle(int x, int y, int radius);
```

Beschreibung *circle* zeichnet einen Kreis mit dem durch *radius* angegebenen Radius um den Mittelpunkt (*x*,*y*). Es wird die aktuelle Zeichenfarbe verwendet.

Der Parameter *linestyle* von *setlinestyle* für die Linienart wirkt sich auf *arc*, *circle*, *ellipse*, *fillellipse*, *sector* und *pieslice* nicht aus; der Parameter *thickness* für die Liniendicke wird dagegen verwendet.

Wenn Kreise nicht ganz rund ausehen sollten, kann dies durch den Aufruf von *setaspectratio* korrigiert werden.

Rückgabewert Keiner.

Siehe auch *arc*, *ellipse*, *fillellipse*, *getaspectratio*, *sector*, *setaspectratio*, *setlinestyle*

cleardevice — graphics.h

Funktion Löscht den Grafikbildschirm.

Syntax

```
void far cleardevice(void);
```

Beschreibung *cleardevice* löscht den gesamten Grafikbildschirm, füllt ihn also mit der aktuellen Hintergrundfarbe und setzt die aktuelle Position (CP) auf den Punkt (0,0) in die obere linke Ecke des Grafikbildschirms.

Rückgabewert Keiner.

Siehe auch *clearviewport*

clearviewport — graphics.h

Funktion Löscht den Inhalt des aktuellen Zeichenfensters.

Syntax

```
void far clearviewport(void);
```

Beschreibung *clearviewport* löscht den Inhalt des aktuellen Zeichenfensters und setzt die aktuelle Position (CP) auf den Ursprung (0,0), relativ zum Zeichenfenster.

Rückgabewert Keiner.

Siehe auch *cleardevice*, *getviewsettings*, *setviewport*

closegraph graphics.h

Funktion Beendet den Grafikmodus.

Syntax

```
void far closegraph(void);
```

Beschreibung *closegraph* gibt den vom Grafikpaket für Treiber, Fonts und interne Puffer belegten Speicherplatz über einen Aufruf von *_graphfreemem* wieder frei und schaltet in den Modus zurück, der vor dem Aufruf von *initgraph* gesetzt war.

Rückgabewert Keiner.

Siehe auch *initgraph*, *setgraphbufsize*

detectgraph graphics.h

Funktion Prüft die Hardware des Computers und bestimmt, welcher Grafiktreiber und welcher Grafikmodus benutzt wird.

Syntax

```
void far detectgraph(int far *graphdriver, int far *graphmode);
```

Beschreibung *detectgraph* prüft den Video-Adapter des Computers und setzt die als *graphdriver* und *graphmode* übergebenen Variablen auf die Kennziffer des passenden Grafiktreibers bzw. den Modus mit der höchstmöglichen Auflösung für diesen Treiber.

Wenn kein grafikfähiger Adapter gefunden werden konnte, bekommt **graphdriver* den Wert -2 (*grNotDetected*), und *graphresult* liefert -2 (*grNotDetected*) zurück.

**graphdriver* ist die Kennziffer des Grafiktreibers (ein Integerwert). In graphics.h sind folgende Konstanten für **graphdriver* definiert:

Tabelle 4.1: Konstanten für *detectgraph*

Name	Wert
DETECT 0	(automatische Erkennung)
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

*graphmode ist ein Integer, der den anfänglichen Grafikmodus festlegt. Wenn *graphdriver auf DETECT gesetzt ist, wird *graphmode automatisch auf die höchstmögliche Auflösung für den erkannten Grafiktreiber gesetzt.

Für *graphmode – den zu setzenden Grafikmodus – sind die folgenden Konstanten in graphics.h definiert:

Tabelle 4.2: Grafiktreiber

Treiber	Grafikmodus	Wert	Spalten * Zeile	Palette	Seiten
CGA	CGAC0	0	320*200	C0	1
	CGAC1	1	320*200	C1	1
	CGAC2	2	320*200	C2	1
	CGAC3	3	320*200	C3	1
	CGAHI	4	640*200	2 Farben	1
MCGA	MCGAC0	0	320*200	C0	1
	MCGAC1	1	320*200	C1	1
	MCGAC2	2	320*200	C2	1
	MCGAC3	3	320*200	C3	1
	MCGAMED	4	640*200	2 Farben	1
	MCGAHI	5	640*480	2 Farben	1
EGA	EGALO	0	640*200	16 Farben	4
	EGAHI	1	640*350	16 Farben	2
EGA64	EGA64LO	0	640*200	16 Farben	1
	EGA64HI	1	640*350	4 Farben	1
EGA-MONO	EGAMONOHI	3	640*350	2 Farben	1*
	EGAMONOHI	3	640*350	2 Farben	2**
HERC	HERCMONOHI	0	720*348	2 Farben	2
ATT400	ATT400C0	0	320*200	C0	1
	ATT400C1	1	320*200	C1	1
	ATT400C2	2	320*200	C2	1
	ATT400C3	3	320*200	C3	1
	ATT400MED	4	640*200	2 Farben	1
	ATT400HI	5	640*400	2 Farben	1
VGA	VGALO	0	640*200	16 Farben	2
	VGAMED	1	640*350	16 Farben	2
	VGAHI	2	640*480	16 Farben	1
PC3270	PC3270HI	0	720*350	2 Farben	1
IBM8514	IBM8514LO	0	640*480	256 Farben	
	IBM8514HI	1	1024*768	256 Farben	

* 64 KByte auf monochromer EGA-Karte
** 256 KByte auf monochromer EGA-Karte

detectgraph wird direkt aufgerufen, um den *initgraph* empfohlenen Grafikmodus zu überschreiben.

Rückgabewert Keiner.

Siehe auch *graphresult*, *initgraph*

drawpoly — graphics.h

Funktion Zeichnet den Umriß eines Polygons.

Syntax

```
void far drawpoly(int numpoints, int far *polypoints);
```

Beschreibung *drawpoly* zeichnet den Umriß eines Polygons mit *numpoints* Eckpunkten in der aktuellen Linienart und -farbe.

Der als *polypoints* übergebene Zeiger muß auf eine Folge von Integerwerten (*numpoints* * 2) zeigen. Jedes Integerpaar steht für die *x*- und *y*-Koordinaten eines Polygonpunktes.

Zum Zeichnen einer geschlossenen Figur mit *n* Eckpunkten muß *polypoints n+1* Koordinatenpaare enthalten, wobei das letzte Paar dieselben Werte wie das erste Paar hat.

Rückgabewert Keiner.

Siehe auch *fillpoly*, *fsloodfill*, *graphresult*, *setwritemode*

ellipse — graphics.h

Funktion Zeichnet einen elliptischen Kreisbogen.

Syntax

```
void far ellipse(int x, int y, int stangle, int endangle,
                 int xradius, int yradius);
```

Beschreibung *ellipse* zeichnet einen elliptischen Kreisbogen um den Mittelpunkt (*x*,*y*) mit den Radien *xradius* (horizontal) und *yradius* (vertikal). *stangle* und *endangle* legen den Anfangs- bzw. Endpunkt des Kreisbogens fest. Hat *x* den Wert 0 und *y* den Wert 360, wird eine vollständige Ellipse gezeichnet. Die Angabe beider Winkel erfolgt in Grad, wobei entgegen dem Uhrzeigersinn gezählt wird (0 Grad = horizontal rechts vom Mittelpunkt, 90 Grad = senkrecht über dem Mittelpunkt usw.).

Der Parameter *linestyle* von *setlinestyle* für die Linienart wirkt sich auf *arc*, *circle*, *ellipse*, *sector* und *pieslice* nicht aus; der Parameter *thickness* für die Liniendicke wird dagegen verwendet.

Rückgabewert Keiner.

Siehe auch *arc*, *circle*, *fillellipse*, *sector*

fillellipse graphics.h

Funktion Zeichnet eine ausgefüllte Ellipse.

Syntax
```
void far fillellipse(int x, int y, int xradius, int yradius);
```

Beschreibung *fillellipse* zeichnet eine Ellipse mit *(x,y)* als Mittelpunkt sowie *xradius* als horizontale und *yradius* als vertikale Achse und füllt sie mit dem aktuellen Füllmuster und der aktuellen Füllfarbe aus.

Rückgabewert Keiner.

Siehe auch *arc*, *circle*, *ellipse*, *pieslice*,

fillpoly graphics.h

Funktion Zeichnet ein ausgefülltes Polygon.

Syntax
```
void far fillpoly(int numpoints, int far *polypoints);
```

Beschreibung *fillpoly* zeichnet den Umriß eines Polygons mit *numpoints* Eckpunkten in der aktuellen Linienart und -farbe (genauso wie *drawpoly*) und füllt es dann mit dem aktuellen Füllmuster und der aktuellen Füllfarbe aus.

polypoints zeigt auf eine Folge von (*numpoints* * 2) Integerwerten. Jedes Integerpaar steht für *x*- und *y*-Koordinaten eines Polygonpunktes.

Rückgabewert Keiner.

Siehe auch *drawpoly*, *floodfill*, *graphresult*, *setfillstyle*

floodfill graphics.h

Funktion Füllt einen umschlossenen Bereich mit dem aktuellen Füllmuster.

Syntax
```
void far floodfill(int x, int y, int border);
```

Beschreibung *floodfill* arbeitet im Gegensatz zu *fillpoly* bit-orientiert; ausgehend vom Punkt (*x,y*) wird der Bildschirm in alle Richtungen mit der aktuellen Füllfarbe im aktuellen Muster gefüllt, wobei das Erreichen eines Pixel mit der Farbe *border* die Operation in der jeweiligen Richtung abbricht.

Wenn sich der Punkt (*x,y*) innerhalb einer Fläche befindet, die mit Linien der Farbe *border* umschlossen ist, füllt *floodfill* diese Fläche. Befindet sich (*x,y*) außerhalb dieser Fläche, dann wird der restliche Bildschirm (bzw. das restliche Zeichenfenster) gefüllt, und die umschlossene Fläche bleibt frei.

Um die Kompatibilität mit späteren Versionen des Compilers zu erhalten, sollte *fillpoly* anstelle von *floodfill* verwendet werden (soweit möglich).

Diese Funktion ist nicht zusammen mit dem IBM 8514-Treiber verwendbar.

Rückgabewert *floodfill* liefert keinen Wert zurück. Wenn während des Flächenfüllens ein Fehler auftritt, liefert ein nachfolgender Aufruf von *graphresult* den Wert -7.

Siehe auch *drawpoly*, *fillpoly*, *graphresult*, *setcolor*, *setfillstyle*

getarccoords graphics.h

Funktion Ermittelt die Koordinaten des letzten Aufrufs von *arc*.

Syntax

```
void far getarccoords(struct arccoordstype far *arccoords);
```

Beschreibung *getarccoords* liefert die Daten des letzten Aufrufs von *arc* in einer Struktur des Typs *arccoordstype* zurück, die in graphics.h folgendermaßen definiert ist:

```
struct arccoordstype {
   int x, y;
   int xstart, ystart, xend, yend;
};
```

Die Elemente dieser Struktur geben den Mittelpunkt (*x*,*y*), den Startpunkt (*xstart*, *ystart*) und den Endpunkt (*xend*, *yend*) des Bogens an. Diese Werte werden benötigt, um einen Kreisbogen z.B. mit einer Linie fortzusetzen.

Rückgabewert Keiner.

Siehe auch *arc*, *fillellipse*, *sector*

getaspectratio graphics.h

Funktion Ermittelt das aktuelle physikalische Höhen-/ Seitenverhältnis des Bildschirms.

Syntax

```
void far getaspectratio(int far *xasp, int far *yasp);
```

Beschreibung Der horizontale Faktor **yasp* ist auf 10000 normiert. Für alle Grafikadapter (mit Ausnahme von VGA) ist **xasp* kleiner als **yasp*, weil die vertikale Ausdehnung der Pixel kleiner als die horizontale ist. Bei VGA-Karten sind die Pixel quadratisch, d.h. **xasp* = **yasp*.

Allgemein kann das Verhältnis zwischen **yasp* und **xasp* folgendermaßen dargestellt werden:

**yasp* = 10000
**xasp* <= 10000

getaspectratio ermittelt die Werte in **xasp* und **yasp*.

Rückgabewert Keiner.

Siehe auch *arc*, *circle*, *ellipse*, *fillellipse*, *pieslice*, *sector*, *setaspectratio*

getbkcolor graphics.h

Funktion Liefert die aktuelle Hintergrundfarbe zurück.

Syntax

```
int far getbkcolor(void);
```

Beschreibung *getbkcolor* liefert die aktuelle Hintergrundfarbe. Details finden Sie unter *setbkcolor*.

Rückgabewert *getbkcolor* liefert die aktuelle Hintergrundfarbe zurück.

Siehe auch *getcolor*, *getmaxcolor*, *getpalette*, *setbkcolor*

getcolor graphics.h

Funktion Liefert die aktuelle Zeichenfarbe zurück.

Syntax

```
int far getcolor(void);
```

Beschreibung Die Zeichenfarbe ist der Wert, auf den Pixel gesetzt werden, wenn Linien und Umrisse gezeichnet werden.

Ein Beispiel: Im Modus CGAC0 enthält die Farbpalette vier Farben: die Hintergrundfarbe, hellgrün, hellrot und gelb. Wenn *getcolor* in diesem Modus den Wert 1 zurückliefert, wird hellgrün als Zeichenfarbe benutzt.

Rückgabewert *getcolor* liefert die aktuelle Zeichenfarbe zurück.

Siehe auch *getbkcolor*, *getmaxcolor*, *getpalette*, *setcolor*

getdefaultpalette graphics.h

Funktion Liefert einen Zeiger auf die Standardfarbpalette zurück.

Syntax

```
struct palettetype *far getdefaultpalette(void);
```

Beschreibung *getdefaultpalette* sucht eine Struktur des Typs *palettetype*. Diese Struktur enthält die Farbpalette, die bei der Initialisierung durch den Treiber mit *initgraph* gesetzt wurde.

Rückgabewert *getdefaultpalette* liefert einen Zeiger auf die Farbpalette zurück, die bei der Initialisierung des aktuellen Treibers gesetzt wurde.

Siehe auch *getpalette*, *initgraph*

getdrivername graphics.h

Funktion Liefert einen Zeiger auf einen String zurück, der den Namen des aktuellen Grafiktreibers enthält.

Syntax

```
char *far getdrivername(void);
```

Beschreibung Nach der Initialisierung des Grafikpakets mit *initgraph* liefert *getdrivername* einen Zeiger auf den Namen des aktuell geladenen Grafiktreibers.

Rückgabewert *getdrivername* liefert einen Zeiger auf einen String zurück, der den Namen des aktuell geladenen Grafiktreibers enthält.

Siehe auch *initgraph*

getfillpattern graphics.h

Funktion Kopiert das benutzerdefinierte Füllmuster in den Speicher.

Syntax

```
void far getfillpattern(char far *pattern);
```

Beschreibung *getfillpattern* kopiert das mit *setfillpattern* vom Benutzer definierte Füllmuster in den 8-Byte-Bereich, auf den *pattern* zeigt.

pattern ist ein Zeiger auf eine Folge von 8 Bytes, bei dem jedes Byte für 8 Pixel des Musters steht. Für jedes Bit, das auf 1 gesetzt ist, wird das korrespondierende Pixel gezeichnet. Zum Beispiel stellt das folgende benutzerdefinierte Füllmuster ein Schachbrett dar:

```
char checkboard[8] = {
   0xAA, 0x55, 0xAA, 0x55, 0xAA, 0xAA, 0x55
};
```

Rückgabewert Keiner.

Siehe auch *getfillsettings*, *setfillpattern*

getfillsettings graphics.h

Funktion Ermittelt Informationen über das aktuelle Füllmuster und die für Flächenfüllungen verwendete Farbe.

Syntax

```
void far getfillsettings(struct fillsettingstype far *fillinfo);
```

Beschreibung

getfillsettings ermittelt das aktuelle Füllmuster und die aktuelle Füllfarbe und speichert diese Informationen in der vom Zeiger *fillinfo* bezeichneten Struktur vom Typ *fillsettingstype*.

Die Struktur des Typs *fillsettingstype* ist in graphics.h folgendermaßen definiert:

```
struct fillsettingstype {
   int pattern;     /* aktuelles Füllmuster */
   int color;       /* aktuelle Füllfarbe */
};
```

Die Funktionen *bar*, *bar3d*, *fillpoly*, *floodfill* und *piesclice* füllen einen Bereich mit dem jeweils aktuellen Füllmuster in der aktuellen Füllfarbe. Es gibt 11 vordefinierte Füllmuster (wie Vollfarbe, Kreuzschraffur, Punkte usw.). Symbolische Namen für die vordefinierten Muster werden durch den Aufzählungstyp *fill_patterns* in graphics.h zur Verfügung gestellt (siehe folgende Tabelle). Zusätzlich können Sie eigene Füllmuster definieren.

Wenn *pattern* gleich 12 (USER_FILL) ist, wird ein benutzerdefiniertes Füllmuster verwendet, ansonsten liefert *pattern* die Nummer eines vordefinierten Musters.

Der Aufzählungstyp *fill_patterns* – definiert in graphics.h – enthält Namen für die vordefinierten Füllmuster sowie einen Indikator für ein benutzerdefiniertes Muster.

Name	Wert	Füllmuster
EMPTY_FILL	0	Vollständig mit der Hintergrundfarbe
SOLID_FILL	1	Vollständig mit der Füllfarbe
LINE_FILL	2	----------
LTSLASH_FILL	3	///
SLASH_FILL	4	/// (dicke Linien)
BKSLASH_FILL	5	\\\ (dicke Linien)
LTBKSLASH_FILL	6	\\\
HATCH_FILL	7	leicht schraffiert
XHATCH_FILL	8	stark schraffiert, überkreuzend
INTERLEAVE_FILL	9	abwechselnde Linien
WIDE_DOT_FILL	10	Punkte, weit auseinander
CLOSE_DOT_FILL	11	Punkte, dicht nebeneinander
USER_FILL	12	benutzerdefiniertes Füllmuster

Alle Füllmuster außer EMPTY_FILL füllen mit der aktuellen Füllfarbe; EMPTY_FILL verwendet die aktuelle Hintergrundfarbe.

Rückgabewert

Keiner.

Siehe auch

getfillpattern, *setfillpattern*, *setfillstyle*

getgraphmode graphics.h

Funktion Ermittelt den aktuellen Grafikmodus.

Syntax

```
int far getgraphmode(void);
```

Beschreibung Vor der Verwendung von *getgraphmode* muß ein erfolgreicher Aufruf von *initgraph* durchgeführt worden sein.

Der Aufzählungstyp *graphics_mode* – definiert in graphics.h – enthält Namen für die vordefinierten Grafikmodi. Eine Tabelle hierzu finden Sie in der Beschreibung von *initgraph*.

Rückgabewert *getgraphmode* liefert die Nummer des durch *initgraph* oder *setgraphmode* gesetzten Grafikmodus zurück.

Siehe auch *getmoderange*, *restorecrtmode*, *setgraphmode*

getimage graphics.h

Funktion Kopiert einen rechteckigen Bildausschnitt in den Speicher.

Syntax

```
void far getimage(int left, int top, int right,
                  int bottom, void far *bitmap);
```

Beschreibung *getimage* kopiert einen Bildausschnitt, dessen obere linke Ecke durch (*left*, *top*) und dessen untere rechte Ecke durch (*right*, *bottom*) festgelegt ist, in die als *bitmap* übergebene Variable. Die ersten beiden 16-Bit-Worte von *bitmap* werden für die Breite und Höhe des kopierten Ausschnitts reserviert, der Bildausschnitt selbst wird im restlichen Teil von *bitmap* gespeichert.

Rückgabewert Keiner.

Siehe auch *imagesize*, *putimage*, *putpixel*

getlinesettings graphics.h

Funktion Ermittelt die aktuelle Linienart, -muster und -dicke.

Syntax

```
void far getlinesettings(struct linesettingstype far *lineinfo);
```

Beschreibung *getlinesettings* ermittelt die aktuelle Linienart, das verwendete Linienmuster sowie die Liniendicke und speichert diese Informationen in der Strukur vom Typ *linesettingstype*, auf die *lineinfo* zeigt.

Der Strukturtyp *linesettingstype* ist in graphics.h folgendermaßen definiert:

```
struct linesettingstype {
   int linestyle;
   unsigned upattern;
   int thickness;
};
```

linestyle gibt die Linienart an, in der im folgenden Linien gezeichnet werden (z.B. durchgezogen, gepunktet, gestrichelt usw.). Der Aufzählungstyp *line_styles*, definiert in graphics.h, stellt Namen für diese Operatoren zur Verfügung:

Name	Wert	Beschreibung
SOLID_LINE	0	durchgezogen
DOTTED_LINE	1	gepunktet
CENTER_LINE	2	Punkt Strich Punkt Strich...
DASHED_LINE	3	gestrichelt
USERBIT_LINE	4	benutzerdefiniert

Für die Dicke einer Linie (*thickness*) sind nur zwei Werte möglich:

Name	Wert	Beschreibung
NORM_WIDTH	1	normal (1 Pixel)
THICK_WIDTH	3	dick (3 Pixel)

upattern ist ein 16-Bit-Muster und hat nur dann eine Wirkung, wenn *linestyle* den Wert USERBIT_LINE (4) hat. In diesem Fall wird ein Pixel in der aktuellen Zeichenfarbe gezeichnet, wenn das korrespondierende Bit auf 1 gesetzt ist. Zum Beispiel ergibt 0xFFFF für *upattern* eine durchgezogene Linie, 0x3333 eine gestrichelte Linie usw.

Auch dann, wenn *setlinestyle* den Wert von *upattern* für *linestyle != USERBIT_LINE* ignoriert, muß dieser Parameter immer angegeben werden.

Rückgabewert Keiner.

Siehe auch *setlinestyle*

getmaxcolor — graphics.h

Funktion Liefert den größten Farbwert zurück, der an die Funktion *setcolor* übergeben werden kann.

Syntax

```
int far getmaxcolor(void);
```

Beschreibung *getmaxcolor* liefert den größten gültigen Farbwert, der für den aktuellen Grafikmodus und -treiber an *setcolor* übergeben werden kann.

Zum Beispiel liefert *getmaxcolor* auf einer EGA-Karte mit 256 KBytes immer 15 zurück, was bedeutet, daß jeder Aufruf von *setcolor* mit einem Wert zwischen 0 und 15 gültig ist. Auf einer CGA-Karte im hochauflösenden Modus oder auf einem Hercules-Monochromadapter liefert *getmaxcolor* den Wert 1.

Rückgabewert *getmaxcolor* liefert den höchsten verfügbaren Farbwert zurück.

Siehe auch *getbkcolor*, *getcolor*, *getpalette*, *getpalettesize*, *setcolor*

getmaxmode graphics.h

Funktion Liefert die höchste Nummer der Grafikmodi des aktuellen Treibers.

Syntax

```
int far getmaxmode(void);
```

Beschreibung *getmaxmode* ermittelt den maximal möglichen Wert für *graphmode* von dem aktuellen Treiber. Diese Funktion ist im Gegensatz zu *getmoderange* auch für Grafiktreiber verwendbar, die nicht von Borland erstellt wurden. Der niedrigste Grafikmodus ist immer 0.

Rückgabewert Zurückgeliefert wird der »höchste« Grafikmodus für den aktuellen Treiber.

Siehe auch *getmodename*, *getmoderange*

getmaxx graphics.h

Funktion Liefert die maximale X-Koordinate des Bildschirms.

Syntax

```
int far getmaxx(void);
```

Beschreibung *getmaxx* liefert in Abhängigkeit vom aktuellen Grafiktreiber und -modus die maximal mögliche X-Koordinate des Bildschirms zurück.

Beispielsweise liefert *getmaxx* auf einem *Color Graphics Adapter* im Modus CGAC0 (320 * 200 Pixel) 319 zurück. *getmaxx* ist besonders geeignet zum Zentrieren oder Festlegen von Ausschnittgrenzen auf dem Bildschirm usw.

Rückgabewert *getmaxx* liefert die maximal mögliche X-Koordinate des Bildschirms zurück.

Siehe auch *getmaxy*, *getx*, *gety*

getmaxy graphics.h

Funktion Liefert die maximal mögliche Y-Koordinate des Bildschirms.

Syntax

```
int far getmaxy(void);
```

Beschreibung *getmaxy* liefert in Abhängigkeit vom aktuellen Grafiktreiber und -modus die maximal mögliche Y-Koordinate des Bildschirms zurück.

Beispielsweise liefert *getmaxy* auf einem *Color Graphics Adapter* im Modus CGAC0 (320 * 200 Pixel) 199 zurück. *getmaxy* ist besonders geeignet zum Zentrieren oder Festlegen von Ausschnittgrenzen auf dem Bildschirm usw.

Rückgabewert *getmaxy* liefert die maximal mögliche Y-Koordinate des Bildschirms zurück.

Siehe auch *getmaxx*, *getx*, *gety*

getmodename graphics.h

Funktion Liefert einen Zeiger auf einen String mit dem Namen des angegebenen Grafikmodus.

Syntax

```
char *far getmodename(int mode_number);
```

Beschreibung *getmodename* erwartet die Nummer eines Grafikmodus als Parameter und liefert einen Zeiger auf den Namen dieses Modus zurück. Die Namen der Grafikmodi sind Bestandteil des jeweiligen Treibers. Die Rückgabewerte (»320x200 CGA P1«, »640x200 CGA« usw.) sind nützlich, um Menüs aufzubauen oder den Status anzuzeigen.

Rückgabewert *getmodename* liefert einen Zeiger auf einen String mit dem Namen des Grafikmodus.

Siehe auch *getmaxmode*, *getmoderange*

getmoderange graphics.h

Funktion Ermittelt den Bereich möglicher Grafikmodi des angegebenen Grafiktreibers.

Syntax

```
void far getmoderange(int graphdriver, int far *lomode,
                      int far *himode);
```

Beschreibung *getmoderange* liefert den niedrigstmöglichen Grafikmodus für den Grafiktreiber *graphdriver* in **lomode* und den höchstmöglichen Modus in **himode*

zurück. Wenn *graphdriver* den Wert eines ungültigen Grafiktreibers enthält, werden **lomode* und **himode* auf -1 gesetzt. Wenn *graphdriver* den Wert -1 enthält, werden die Modi des aktuell geladenen Treibers übergeben.

Rückgabewert Keiner.

Siehe auch *getgraphmode*, *getmaxmode*, *getmodename*, *initgraph*, *setgraphmode*

getpalette graphics.h

Funktion Liefert Informationen über die aktuelle Farbpalette.

Syntax

```
void far getpalette(struct palettetype far *palette);
```

Beschreibung *getpalette* liefert die Farben der aktuellen Farbpalette und ihre Gesamtgröße in der Struktur vom Typ *palettetype* zurück, auf die *palette* zeigt.

Die Konstante MAXCOLORS und die Struktur *palettetype* sind in graphics.h folgendermaßen definiert:

```
#define MAXCOLORS 15

struct palettetype {
   unsigned char size;
   signed char colors[MAXCOLORS+1];
};
```

Das Element *size* enthält die Anzahl der tatsächlich benutzten Einträge des Array *colors*, dessen Wert vom aktiven Grafikmodus und dem verwendeten Treiber abhängig ist.

Die Elemente von *colors* (ein Array mit *size* Bytes) enthalten die aktuellen Farbnummern für jeden Eintrag in der Palette.

getpalette kann nicht zusammen mit dem IBM-8514-Adapter verwendet werden.

Rückgabewert Keiner.

Siehe auch *getbkcolor*, *getcolor*, *getdefaultpalette*, *getmaxcolor*, *setallpalette*, *setpalette*

getpalettesize graphics.h

Funktion Liefert die Größe der aktuellen Farbpalette zurück.

Syntax

```
int far getpalettesize(void);
```

Beschreibung Mit *getpalettesize* läßt sich ermitteln, wieviele Einträge in der Farbpalette im aktuellen Grafikmodus gesetzt werden können. Beispielsweise wird für eine EGA-Karte im Farbmodus 16 zurückgeliefert.

Rückgabewert *getpalettesize* liefert die Anzahl der Einträge in der aktuellen Farbpalette zurück.

Siehe auch *setpalette*, *setallpalette*

getpixel graphics.h

Funktion Ermittelt die Farbe eines Pixel.

Syntax

```
unsigned far getpixel(int x, int y);
```

Beschreibung *getpixel* liefert die Farbe des Pixel auf dem durch (*x*,*y*) angegebenen Punkt zurück.

Rückgabewert *getpixel* liefert die Farbe des angegebenen Pixel zurück.

Siehe auch *getimage*, *putpixel*

gettextsettings graphics.h

Funktion Ermittelt Informationen über den aktuellen Grafik-Font.

Syntax

```
void far gettextsettings(struct textsettingstype far *texttypeinfo);
```

Beschreibung *gettextsettings* ermittelt die aktuell gesetzten Parameter über den aktuellen Font, die Richtung (horizontal oder vertikal), den Vergrößerungsfaktor und die Ausrichtung und liefert sie in der über *texttypeinfo* übergebenen Struktur vom Typ *textsettingstype* zurück.

Der Strukturtyp *textsettingstype* ist in graphics.h folgendermaßen definiert:

```
struct textsettingstype {
  int font;      /* Nummer des aktiven Zeichensatzes */
  int direction; /* horizontal oder vertikal */
  int charsize;  /* Vergrößerungsfaktor */
  int horiz;     /* horizontale Ausrichtung */
  int vert;      /* vertikale Ausrichtung */
};
```

Eine Beschreibung der Felder finden Sie in der Beschreibung von *settextstyle*.

Rückgabewert Keiner.

Siehe auch *outtext, outtextxy, registerbgifont, settextjustify, settextstyle, setusercharsize, textheigth, textwidth*

getviewsettings graphics.h

Funktion Ermittelt Informationen über das aktuelle Zeichenfenster.

Syntax

```
void far getviewsettings(struct viewporttype far *viewport);
```

Beschreibung *getviewsettings* speichert in den Elementen der Struktur vom Typ *viewporttype*, auf die *viewport* zeigt, Informationen über das aktuelle Zeichenfenster.

Der Strukturtyp *viewporttype* ist in graphics.h folgendermaßen definiert:

```
struct viewporttype {
   int left, top,    /* obere linke und     */
   right, bottom;    /* untere rechte Ecke  */
   int clip;         /* Clip-Funktion an/aus */
};
```

Rückgabewert Keiner.

Siehe auch *clearviewport, getx, gety, setviewport*

getx graphics.h

Funktion Liefert die X-Koordinate der aktuellen Position (CP) zurück.

Syntax

```
int far getx(void);
```

Beschreibung *getx* liefert die X-Koordinate der aktuellen Grafikposition (CP).

Die von *getx* zurückgelieferten Koordinaten sind relativ zum aktuellen Zeichenfenster.

Rückgabewert *getx* liefert die X-Koordinate der aktuellen Position (CP).

Siehe auch *getmaxx, getmaxy, getviewsettings, gety, moveto*

gety graphics.h

Funktion Liefert die Y-Koordinate der aktuellen Position (CP) zurück.

Syntax

```
int far gety(void);
```

Beschreibung *gety* liefert die Y-Koordinate der aktuellen Grafikposition (CP).

Die von *gety* zurückgelieferten Koordinaten sind relativ zum aktuellen Zeichenfenster.

Rückgabewert *gety* liefert die Y-Koordinate der aktuellen Position (CP) zurück.

Siehe auch *getmaxx*, *getmaxy*, *getviewsettings*, *getx*, *moveto*

graphdefaults graphics.h

Funktion Setzt sämtliche Einstellungen des Grafikpakets auf die Voreinstellungen zurück.

Syntax

```
void far graphdefaults(void);
```

Beschreibung *graphdefaults* setzt alle Einstellungen des Grafikpakets auf die Voreinstellungen:

- setzt das Zeichenfenster auf die vollständige Bildschirmgröße;
- positioniert die aktuelle Position (CP) auf (0,0);
- setzt die voreingestellte Hintergrundfarbe, Zeichenfarbe und Farbpalette;
- setzt die voreingestellte Art und das voreingestellte Muster für Flächenfüllungen;
- setzt den voreingestellten Font und die voreingestellte Ausrichtung.

Rückgabewert Keiner.

Siehe auch *initgraph*, *setgraphmode*

grapherrormsg graphics.h

Funktion Liefert einen Zeiger auf eine Fehlermeldung des Grafikpakets.

Syntax

```
char *far grapherrormsg(int errorcode);
```

Beschreibung *grapherrormsg* liefert einen Zeiger auf einen String zurück, der den Text der zu *errorcode* gehörigen Fehlermeldung enthält. Für *errorcode* wird der von *graphresult* gelieferte Wert benutzt.

In Kapitel 4 des *Referenzhandbuchs* finden Sie unter *errno* eine Liste aller möglichen Fehler und ihrer Kurzbezeichnungen.

Rückgabewert *grapherrormsg* liefert einen Zeiger auf einen String mit dem Text einer Fehlermeldung.

Siehe auch *graphresult*

_graphfreemem graphics.h

Funktion Vom Grafikpaket verwendete Routine zur Freigabe von dynamisch belegtem Speicherplatz.

Syntax

```
void far _graphfreemem(void far *ptr, unsigned size);
```

Beschreibung Die Grafikbibliothek ruft *_graphfreemem* auf, um einen vorher durch *_graphgetmem* belegten Bereich wieder freizugeben. Es ist möglich, *_graphfreemem* durch eine eigene Version zu ersetzen (»Überladen der Funktion«): damit hat man die Kontrolle über die Speicherverwaltung der Grafik. Die Standardversion dieser Routine ruft nur *free* auf.

Die Deklaration Ihrer Version von *_graphfreemem* muß exakt in der unter »Syntax« gezeigten Form geschehen.

Rückgabewert Keiner.

Siehe auch *_graphgetmem*, *setgraphbufsize*

_graphgetmem graphics.h

Funktion Vom Grafikpaket verwendete Routine zur dynamischen Belegung von Speicherplatz.

Syntax

```
void far *far _graphgetmem(unsigned size);
```

Beschreibung Die Routinen der Grafikbibliothek (nicht das Benutzerprogramm) rufen *_graphgetmem* auf, um Speicher für interne Puffer, Grafiktreiber und Schriften zu belegen.

Es ist möglich, *_graphgetmem* durch eine eigene Version zu ersetzen: damit hat man die Kontrolle über die Speicherverwaltung der Grafik. Die Standardversion dieser Routine ruft lediglich *malloc* auf.

Die Deklaration Ihrer Version von *_graphgetmem* muß exakt in der unter »Syntax« gezeigten Form geschehen.

Rückgabewert Keiner.

Siehe auch *_graphfreemem*, *initgraph*, *setgraphbufsize*

graphresult graphics.h

Funktion Liefert den Fehlercode der letzten fehlgeschlagenen Grafikoperation zurück.

Syntax

```
int far graphresult(void);
```

Beschreibung

graphresult liefert den Fehlercode der letzten Grafikoperation, die einen Fehler erzeugt hat, und setzt den Fehlerstatus auf *grOk* zurück.

Die folgende Tabelle enthält eine Auflistung der Fehlernummern und Meldungen, die von *graphresult* zurückgeliefert werden. Der Aufzählungstyp *graph_errors*, der in graphics.h deklariert ist, definiert die Werte in dieser Tabelle.

Name	Wert	Text/Beschreibung
grOk	0	*No error* Fehlerfreie Ausführung
grNoInitGraph	-1	*(BGI) graphics not installed (use initgraph)* Grafiktreiber (.BGI) nicht geladen – *initgraph* muß zuerst aufgerufen werden
grNotDetected	-2	*Graphics hardware not detected* Kein grafikfähiger Adapter gefunden
grFileNotFound	-3	*Device driver file not found* Grafiktreiberdatei nicht gefunden
grInvalidDriver	-4	*Invalid device driver file* Grafiktreiberdatei defekt
grNoLoadMem	-5	*Not enough memory to load driver* Nicht genug Platz im Hauptspeicher für den Grafiktreiber
grNoScanMem	-6	*Out of memory in scan fill* Nicht genug Platz im Hauptspeicher (*bar*, *fillpoly*)
grNoFloodMem	-7	*Out of memory in flood fill* Nicht genug Platz im Hauptspeicher (*floodfill*)
grFontNotFound	-8	*Font file not found* Font-Datei nicht gefunden
grNoFontMem	-9	*Not enough memory to load font* Nicht genug Platz im Hauptspeicher für den Font
grInvalidMode	-10	*Invalid Graphics mode for selected driver* Grafikmodus wird vom gewählten Treiber nicht unterstützt
grError	-11	*Graphics Error* Nicht näher bestimmbarer Fehler
grIOerror	-12	*Graphics I/O error* Ein-/Ausgabefehler
grInvalidFont	-13	*Invalid font file* Font-Datei ungültig
grInvalidFontNum	-14	*Invalid font number* Kennziffer für Font nicht definiert
grInvalidDeviceNum	-15	*Invalid device number* Kennziffer für Grafiktreiber nicht definiert
grInvalidVersion	-18	*Invalid version number* Version wird nicht unterstützt

Der Aufruf von *graphresult* setzt den Statuscode automatisch auf 0 zurück. Prüfungen des Statuscodes sollten deshalb über eine temporäre Variable erfolgen.

Rückgabewert *graphresult* liefert die aktuelle Fehlernummer des Grafikpakets (mögliche Werte von -15 bis 0) zurück, *grapherrormsg* einen Zeiger auf einen String mit dem Text der von *graphresult* gelieferten Meldung.

Siehe auch *detectgraph*, *drawpoly*, *fillpoly*, *floodfill*, *grapherrormsg*, *initgraph*, *pieslice*, *registerbgidriver*, *registerbgifont*, *setallpalette*, *setcolor*, *setfillstyle*, *setgraphmode*, *seltlinestyle*, *setpalette*, *settextjustify*, *settextstyle*, *setusercharsize*, *setviewport*, *setvisualpage*

imagesize graphics.h

Funktion Berechnet die Anzahl der Bytes, die zur Speicherung grafischer Bildausschnitte benötigt werden.

Syntax

```
unsigned far imagesize(int left, int top, int right, int bottom);
```

Beschreibung *imagesize* berechnet die Anzahl von Bytes, die zur Speicherung eines Bildausschnitts benötigt werden. Ist die zur Speicherung benötigte Anzahl von Bytes größer oder gleich *64 KByte - 1 Byte*, liefert *imagesize* den Wert 0xFFFF (-1) zurück.

Rückgabewert *imagesize* liefert die Größe des angegebenen Bildausschnitts in Bytes zurück.

Siehe auch *getimage*, *putimage*

initgraph graphics.h

Funktion Initialisiert das Grafiksystem.

Syntax

```
void far initgraph(int far *graphdriver, int far *graphmode,
                   char far *pathtodriver);
```

Beschreibung Die Funktion *initgraph* initialisiert das Grafiksystem, indem sie einen Grafiktreiber von der Platte lädt (bzw. einen mit *registerbgidriver* eingebundenen Treiber überprüft) und in den Grafikmodus schaltet. Um das Grafiksystem zu starten, rufen Sie zuerst *initgraph* auf. *initgraph* lädt den Grafiktreiber und schaltet das System in den Grafikmodus, den Sie angeben.

Sie können *initgraph* auch anweisen, die angeschlossene Grafikkarte zu untersuchen und einen entsprechenden Treiber auszuwählen.

Falls Sie *initgraph* zur automatischen Erkennung veranlassen, werden Grafiktreiber und Grafikmodus durch einen Aufruf von *detectgraph* ausgewählt. *initgraph* wählt für alle Einstellungen des Grafiksystems die Voreinstellungen (aktuelle Position (CP), Farbpalette, Farbe, Zeichenfenster usw.) und setzt *graphresult* auf 0. Normalerweise lädt *initgraph* einen Gra-

fiktreiber, indem über einen Aufruf von *_graphgetmem* Speicher für den Treiber belegt und die entsprechende .BGI-Datei von der Platte geladen wird. Alternativ zu diesem dynamischen Verfahren können Sie auch eine oder mehrere Grafiktreiberdateien direkt in die ausführbare Datei einbinden.

pathtodriver ist ein Zeiger auf einen String, der den Suchweg für Grafiktreiberdateien festlegt: *initgraph* sucht zuerst im angegebenen Verzeichnis, danach ggf. im aktuellen Verzeichnis. Wenn *pathtodriver* NULL ist, findet die Suche nur im aktuellen Verzeichnis statt. Dieser Suchweg bestimmt auch das Verzeichnis, in dem *settextstyle* nach Vektor-Font-Dateien (*.CHR) sucht.

Für **graphdriver* – die Kennziffer des Grafiktreibers – sind die folgenden Konstanten des Aufzählungstyps *graphics_drivers* in graphics.h definiert:

Tabelle 4.3: Grafiktreiber-Konstanten

Name	Wert
DETECT	0 (automatische Erkennung)
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

**graphmode* ist ein Integer, der den Grafikmodus für die Initialisierung festlegt (außer, wenn für **graphdriver* DETECT übergeben wird; in diesem Fall wählt *initgraph* die höchstmögliche Auflösung für den gefundenen Treiber). Sie können **graphmode* einen Wert des Aufzählungstyps *graphics_modes* (definiert in graphics.h und aufgeführt in Tabelle 4.5) zuweisen.

graphdriver und *graphmode* müssen auf gültige, in den Tabellen 4.3 und 4.5 aufgeführte Werte gesetzt werden. Andernfalls sind die Ergebnisse unvorhersehbar. Ausnahme: Sie weisen *graphdriver* den Wert DETECT zu.

In den Farbpaletten C0 bis C3 (d.h. den CGA-Karten und Kompatiblen) kann nur die Farbe des Hintergrundes (Eintrag 0) verändert werden. Die restlichen Farben sind durch die jeweilige Palette festgelegt. Diese Paletten sind detailliert im Kapitel 3 beschrieben und in der folgenden Tabelle 4.4 zusammengefaßt:

Tabelle 4.4: Farbpaletten

Palette	Pixelwert		
	1	2	3
C0	hellgrün	hellrot	gelb
C1	helltürkis	hell-violett	weiß
C2	grün	rot	braun
C3	türkis	violett	Hellgrau

Nach einem Aufruf von *initgraph* wird **graphdriver* auf den aktuellen Grafiktreiber und **graphmode* auf den aktuellen Grafikmodus gesetzt.

Für *graphmode* – den zu setzenden Grafikmodus – sind die folgenden Konstanten in graphics.h definiert:

Tabelle 4.5: Grafikmodi

Treiber	Grafikmodus	Wert	Spalten * Zeilen	Palette	Seiten
CGA	CGAC0	0	320*200	C0	1
	CGAC1	1	320*200	C1	1
	CGAC2	2	320*200	C2	1
	CGAC3	3	320*200	C3	1
	CGAHI	4	640*200	2 Farben	1
MCGA	MCGAC0	0	320*200	C0	1
	MCGAC1	1	320*200	C1	1
	MCGAC2	2	320*200	C2	1
	MCGAC3	3	320*200	C3	1
	MCGAMED	4	640*200	2 Farben	1
	MCGAHI	5	640*480	2 Farben	1
EGA	EGALO	0	640*200	16 Farben	4
	EGAHI	1	640*350	16 Farben	2
EGA64	EGA64LO	0	640*200	16 Farben	1
EGA64HI		1	640*350	4 Farben	1
EGAMONO	EGAMONOHI	3	640*350	2 Farben	1*
	EGAMONOHI	3	640*350	2 Farben	2**
HERC	HERCMONOHI	0	720*348	2 Farben	2
ATT400	ATT400C0	0	320*200	C0	1
	ATT400C1	1	320*200	C1	1
	ATT400C2	2	320*200	C2	1
	ATT400C3	3	320*200	C3	1
	ATT400MED	4	640*200	2 Farben	1
	ATT400HI	5	640*400	2 Farben	1
VGA	VGALO	0	640*200	16 Farben	2
	VGAMED	1	640*350	16 Farben	2
	VGAHI	2	640*480	16 Farben	1
PC3270	PC3270HI	0	720*350	2 Farben	1

IBM8514	IBM8514LO	0	640*480	256 Farben
	IBM8514HI	1	1024*768	256 Farben

* 64 KByte auf monochroner EGA-Karte
** 256 KByte auf monochromer EGA-Karte

Rückgabewert

initgraph setzt immer den internen Fehlercode. Bei erfolgreicher Ausführung wird der Code auf 0 gesetzt. Wenn *initgraph* nicht fehlerfrei ausgeführt werden kann, bekommt **graphdriver* einen der folgenden Werte zugewiesen, die auch von *graphresult* zurückgeliefert werden:

Name	Wert	Text/Beschreibung
grNotDetected	-2	Kein grafikfähiger Adapter gefunden
grFileNotFound	-3	Grafiktreiberdatei nicht gefunden
grInvalidDriver	-4	Grafiktreiberdatei ungültig
grNoLoadMem	-5	Nicht genug Platz im Hauptspeicher für den Grafiktreiber

Siehe auch

closegraph, *detectgraph*, *getdefaultpalette*, *getdrivername*, *getgraphmode*, *getmoderange*, *graphdefaults*, *_graphgetmem*, *graphresult*, *installuserdriver*, *registerbgidriver*, *registerbgifont*, *restorecrtmode*, *setgraphbufsize*, *setgraphmode*

installuserdriver — graphics.h

Funktion

Fügt einen neuen Gerätetreiber in die BGI-Gerätetreibertabelle ein.

Syntax

```
int far installuserdriver(char far *name, int huge (*detect)(void));
```

Beschreibung

Mit *installuserdriver* können Sie einen neu erworbenen Gerätetreiber in die interne BGI-Tabelle einfügen. Der Parameter *name* gibt den Namen der neuen Gerätetreiberdatei an (.BGI). *detect* steht für einen Zeiger auf eine optionale automatische Erkennungsfunktion, die zu dem neuen Treiber mitgeliefert sein kann. Diese Erkennungsfunktion hat keine Parameter und liefert einen Integerwert zurück.

Es gibt zwei Möglichkeiten für den Einsatz des neuen Treibers. Nehmen wir an, Sie haben eine neue Grafikkarte mit dem Namen *Spiffy Graphics Array (SGA)* erworben, und der Hersteller liefert dazu einen BGI-Gerätetreiber (SGA.BGI). Sie können nun ganz einfach diesen Treiber durch Aufruf von *installuserdriver* installieren und den Rückgabewert (die zugewiesene Treibernummer) direkt an *initgraph* übergeben.

Die zweite (allgemeinere) Möglichkeit, den Treiber zu benutzen, besteht darin, eine automatische, von *initgraph* als Bestandteil der Hardware-Erkennungsroutine aufzurufende, Erkennungsfunktion einzubinden. Voraussetzung ist, daß der Hersteller der SGA diese automatische Erkennungsfunktion mitliefert. Wenn Sie durch Aufruf von *installuserdriver* den

Treiber installieren, übergeben Sie die Adresse dieser Funktion zusammen mit dem Namen der Gerätetreiberdatei.

Nach der Übergabe des Gerätetreiberdateinamens und der automatischen SGA-Erkennungsfunktion rufen Sie *initgraph* auf und lassen den automatischen Erkennungsprozeß ablaufen. Bevor *initgraph* seine eigene automatische Erkennungsfunktion (*detectgraph*) aufruft, wird zuerst die SGA-Erkennungsfunktion aufgerufen. Kann die automatische SGA-Erkennungsfunktion die SGA-Hardware nicht finden, liefert sie den Wert -11 (grError) zurück, und *initgraph* führt dann den normalen Hardware-Erkennungsprozeß durch, der den Aufruf weiterer vom Hersteller gelieferter Erkennungsfunktionen beinhalten kann. Weitere vorhandene Funktionen werden in der Reihenfolge aufgerufen, in der sie »installiert« wurden. Stellt die automatische SGA-Erkennungsfunktion jedoch fest, daß eine SGA-Karte vorhanden ist, liefert sie eine positive Modusnummer zurück. *initgraph* sucht und lädt SGA.BGI, setzt die Hardware in den voreingestellten, von der Funktion empfohlenen Grafikmodus und übergibt die Steuerung wieder an Ihr Programm.

Sie können bis zu 10 Treiber gleichzeitig installieren.

Rückgabewert *installuserdriver* liefert den Integer-Wert zurück, den Sie *initgraph* übergeben müssen, wenn Sie den neu installierten Grafiktreiber manuell auswählen wollen.

Siehe auch *initgraph*, *registerbgidriver*

installuserfont graphics.h

Funktion Zur Installation von Vektor-Fonts (*.CHR), die nicht direkt von Borland angeboten werden.

Syntax

```
int far installuserfont(char far *name);
```

Beschreibung *name* zeigt den Namen einer Vektor-Font Datei im aktuellen Verzeichnis (Pfadnamen werden nicht unterstützt). Mit *installuserfont* können maximal zwanzig Fonts in einem Programm installiert werden.

Rückgabewert *installuserfont* liefert einen Integer zurück, der im weiteren Programmverlauf als Kennziffer für den entsprechenden Font interpretiert wird und an *settextstyle* übergeben werden kann. Wenn die Font-Tabelle des Grafiksystems keinen weiteren Eintrag aufnehmen kann, liefert die Funktion *grError* den Wert -11 zurück.

Siehe auch settextstyle

line — graphics.h

Funktion Zeichnet eine Linie zwischen zwei Punkten.

Syntax

```
void far line(int x1, int y1, int x2, int y2);
```

Beschreibung Die Funktion verwendet die mit *setlinestyle* gesetzte Linienart und -dicke sowie die aktuell gesetzte Farbe.

line zeichnet eine Linie vom Punkt (*x1,y1*) zum Punkt (*x2,y2*), ohne die aktuelle Position (CP) zu verändern.

Rückgabewert Keiner.

Siehe auch *getlinesettings*, *linerel*, *lineto*, *setcolor*, *setlinestyle*, *setwritemode*

linerel — graphics.h

Funktion Zeichnet eine Linie relativ zur aktuellen Position (CP).

Syntax

```
void far linerel(int dx, int dy);
```

Beschreibung *linerel* interpretiert die als *dx,dy* übergebenen Werte als relative Entfernung zur aktuellen Position (CP) und zeichnet eine Linie von der aktuellen Position zum durch (*dx,dy*) angegebenen Punkt. Die aktuelle Position (CP) wird dadurch um (*dx,dy*) bewegt.

Rückgabewert Keiner.

Siehe auch *getlinesettings*, *line*, *lineto*, *setcolor*, *setlinestyle*, *setwritemode*

lineto — graphics.h

Funktion Zeichnet eine Linie von der aktuellen Position (CP) zum angegebenen Punkt.

Syntax

```
void far lineto(int x, int y);
```

Beschreibung *lineto* zeichnet eine Linie von der aktuellen Position (CP) zum Punkt (*x,y*). Die aktuelle Position ist danach der Punkt (*x,y*).

Rückgabewert Keiner.

Siehe auch *getlinesettings*, *line*, *linerel*, *setcolor*, *setlinestyle*, *setvisualpage*, *setwritemode*

moverel — graphics.h

Funktion Bewegt die aktuelle Position (CP) um eine relative Entfernung.

Syntax

```
void far moverel(int dx, int dy);
```

Beschreibung *moverel* bewegt die aktuelle Position (CP) um *dx* Pixel in der X-Richtung und um *dy* Pixel in der Y-Richtung.

Rückgabewert Keiner.

Siehe auch *moveto*

moveto — graphics.h

Funktion Setzt die aktuelle Position (CP) auf einen bestimmten Punkt.

Syntax

```
void far moveto(int x, int y);
```

Beschreibung *moveto* setzt die aktuelle Position (CP) auf den Punkt (*x*,*y*) im Zeichenfenster.

Rückgabewert Keiner.

Siehe auch *moverel*

outtext — graphics.h

Funktion Gibt Text ab der aktuellen Position (CP) aus.

Syntax

```
void far outtext(char far *textstring);
```

Beschreibung *outtext* gibt den als *textstring* übergebenen String ab der aktuellen Position (CP) im Zeichenfenster aus. *outtext* benutzt die Schriftart und Formatierung, die durch *settextstyle* (Schriftart, Rotation, Größe) und *settextjustify* (linksbündig, zentriert, rechtsbündig) vorgegeben ist.

Falls die horizontale Justierung LEFT_TEXT ist und die Textausgabe in horizontaler Richtung erfolgt (HORIZ_DIR), wird die X-Koordinate der aktuellen Position um *textwidth(textstring)* geändert, ansonsten bleibt sie unverändert.

Damit das Programm bei einem Wechsel der Schriftart und -größe nicht modifiziert werden muß, sollte der Platzbedarf auszugebender Strings immer über *textwidth* und *textheight* berechnet werden.

Ein automatischer Umbruch wie im Textmodus findet nicht statt: Ausgaben, die über den rechten Rand des Bildschirms bzw. des aktuellen Zeichenfensters hinausgehen würden, schneidet *outtext* einfach ab.

outtext funktioniert nur im Grafik-, nicht aber im Textmodus.

Rückgabewert Keiner.

Siehe auch *gettextsettings*, *outtextxy*, *settextjustify*, *textheight*, *textwidth*

outtextxy graphics.h

Funktion Gibt Text ab einem bestimmten Punkt aus.

Syntax

```
void far outtextxy(int x, int y, char far *textstring);
```

Beschreibung *outtextxy* gibt den als *textstring* übergebenen String ab der Position (*x*,*y*) im Zeichenfenster aus. Die aktuelle Position wird dabei nicht verändert.

outtextxy benutzt die Schriftart und Formatierung, die durch *settextstyle* (Schriftart, Rotation, Größe) und *settextjustify* (linksbündig, zentriert, rechtsbündig) vorgegeben ist.

Damit das Programm bei einem Wechsel der Schriftart und -größe nicht modifiziert werden muß, sollte der Platzbedarf auszugebender Strings immer über *textwidth* und *textheight* berechnet werden.

Ein automatischer Umbruch wie im Textmodus findet nicht statt: Ausgaben, die über den rechten Rand des Bildschirms bzw. des aktuellen Zeichenfensters hinausgehen würden, schneidet *outtextxy* einfach ab.

outtextxy funktioniert nur im Grafik-, nicht aber im Textmodus.

Rückgabewert Keiner.

Siehe auch *gettextsettings*, *outtext*, *textheight*, *textwidth*

pieslice graphics.h

Funktion Zeichnet ein ausgefülltes »Kuchenstück«.

Syntax

```
void far pieslice(int x, int y, int stangle, int endangle, int radius);
```

Beschreibung *pieslice* zeichnet in der aktuellen Zeichenfarbe ein Kreisbogenstück mit dem Mittelpunkt (*x*,*y*), wobei *stangle* den Startpunkt und *endangle* den Endpunkt des mit *radius* gezogenen Bogens festlegen.

Die Endpunkte des Kreisbogens werden mit dem Kreismittelpunkt verbunden, danach füllt *pieslice* das entstandene Kuchenstück mit dem durch *setfillpattern* bzw. *setfillstyle* festgelegten Muster aus.

Die Angabe beider Winkel erfolgt in Grad, wobei entgegen dem Uhrzeigersinn gezählt wird (0 Grad = horizontal rechts vom Mittelpunkt, 90 Grad = senkrecht über dem Mittelpunkt usw.).

Wenn Ihr System mit einem CGA- oder Monochromadapter ausgestattet ist, werden einige der Beispiele zur Verwendung der Grafikfunktionen nicht einwandfrei funktionieren. Sie sollten dann den Wert 1 (eins) statt der symbolischen Farbkonstanten an die Funktionen übergeben, die die Füll- oder Zeichenfarbe setzen (zum Beispiel *setcolor*, *setfillstyle* oder *setlinestyle*).

Rückgabewert Keiner.

Siehe auch *fillellipse*, *fill_patterns* (Aufzählungstyp), *graphresult*, *sector*, *setfillstyle*

putimage graphics.h

Funktion Kopiert den Inhalt einer Puffervariablen bitweise in einen rechteckigen Bildausschnitt.

Syntax

```
void far putimage(int left, int top, void far *bitmap, int op);
```

Beschreibung *putimage* gibt den zuvor mit *getimage* gespeicherten Bildausschnitt auf den Bildschirm aus, wobei (*left*,*top*) die obere linke Ecke des »Zielbereichs« festlegt. *bitmap* zeigt auf den Speicherbereich, in dem der Bildausschnitt gespeichert ist.

Der Parameter *op* steuert, wie die Farbe jedes Pixel im »Zielbereich« basierend auf der Farbe des bereits dort vorhandenen Pixel und der des Pixel im Speicher (Quellbereich) berechnet wird.

Für den Parameter *op* sind in graphics.h die folgenden Namen für Operatoren definiert:

Name	Wert	Beschreibung
COPY_PUT	0	Kopie (Überschreiben des »Zielbereichs«)
XOR_PUT	1	Exklusives-ODER-Operation mit dem »Zielbereich«
OR_PUT	2	ODER-Operation
AND_PUT	3	UND-Operation
NOT_PUT	4	Quellbereich invertieren

Mit anderen Worten kopiert COPY_PUT das Quell-Bitmap-Bild auf den Bildschirm, XOR_PUT ver-XOR-t das Quellbild mit dem auf dem Bildschirm bereits befindlichen Bild, OR_PUT ver-ODER-t die beiden Bilder usw.

Rückgabewert Keiner.

Siehe auch *getimage*, *imagesize*, *putpixel*, *setvisualpage*

putpixel graphics.h

Funktion Zeichnet einen Punkt auf die angegebenen Koordinaten.

Syntax
```
void far putpixel(int x, int y, int color);
```

Beschreibung *putpixel* zeichnet einen Punkt mit der Farbe *color* auf dem Punkt (*x*,*y*).

Rückgabewert Keiner.

Siehe auch *getpixel*, *putimage*

rectangle graphics.h

Funktion Zeichnet ein Rechteck.

Syntax
```
void far rectangle(int left, int top, int right, int bottom);
```

Beschreibung *rectangle* zeichnet ein Rechteck, dessen obere linke Ecke durch (*left*, *top*) und dessen untere rechte Ecke durch (*right*, *bottom*) festgelegt ist, wobei die durch *setlinestyle* gesetzten Parameter (Linienart, -dicke und -farbe) verwendet werden.

Rückgabewert Keiner.

Siehe auch *bar*, *bar3d*, *setcolor*, *setlinestyle*

registerbgidriver graphics.h

Funktion Registriert einen eingebundenen oder geladenen Grafiktreiber.

Syntax
```
int registerbgidriver(void (*driver)(void));
```

Beschreibung *registerbgidriver* ermöglicht es einem Benutzer, eine Treiberdatei zu laden und den Treiber zu »registrieren«. Sobald dessen Speicheradresse an *registerbgidriver* übergeben wurde, verwendet *initgraph* den somit registrierten Treiber. Ein benutzerregistrierter Treiber kann von einem Datenträger auf den Heap geladen werden. Er kann aber auch mit BINOBJ.EXE in eine .OBJ-Datei konvertiert und in die .EXE-Datei gelinkt werden.

Der Aufruf von *registerbgidriver* informiert das Grafiksystem darüber, daß der Treiber, auf den *driver* zeigt, zur Linkzeit eingebunden wurde. Diese Routine überprüft den hinzugelinkten Code des angegebenen Treibers; ist er gültig, so wird er in internen Tabellen registriert.

Durch die Verwendung des Namens eines hinzugelinkten Treibers in einem Aufruf von *registerbgidriver* weisen Sie außerdem den Compiler (und Linker) an, die .OBJ-Datei mit diesem öffentlichen Namen hinzuzulinken.

Rückgabewert Nach einer fehlerfreien Ausführung liefert *registerbgidriver* die Kennziffer des »registrierten« Grafiktreibers zurück. Negative Funktionsergebnisse entsprechen den Statuscodes des Grafikpakets und zeigen einen Fehler an.

Wenn ein vom Programmierer erstellter Treiber registriert wird, *muß* das Ergebnis von *registerbgidriver* als Treibernummer an *initgraph* übergeben werden.

Siehe auch *graphresult*, *initgraph*, *installuserdriver*, *registerbgifont*

registerbgifont graphics.h

Funktion Registiert einen als .OBJ-Datei eingebundenen Vektor-Font.

Syntax

```
int registerbgifont(void (*font)(void));
```

Beschreibung *registerbgifont* erwartet die Adresse einer als .OBJ-Datei in das Programm eingebundenen Vektorschrift und trägt diesen Font nach einer Prüfung in eine interne Tabelle für *settextstyle* ein. Auf diese Weise »registrierte« Fonts werden nicht mehr als .CHR-Dateien auf der Diskette gesucht und auf den Heap geladen, sondern direkt verwendet.

Da Fonts nur indirekt von anderen Routinen angesprochen werden, erfüllt der Aufruf der »Registrierfunktion« einen weiteren Zweck: er teilt dem Linker von Borland C++ mit, daß die entsprechende .OBJ-Datei mit in das Programm aufgenommen werden muß.

Wenn ein vom Programmierer erstellter Font registriert wird, *muß* das Ergebnis von *registerbgifont* als Fontnummer an *settextstyle* übergeben werden.

Rückgabewert Nach einer fehlerfreien Ausführung liefert *registerbgifont* die Nummer des »registrierten« Grafik-Fonts zurück. Negative Funktionsergebnisse entsprechen den Statuscodes des Grafikpakets und zeigen einen Fehler an.

Siehe auch *graphresult*, *initgraph*, *installuserdriver*, *registerbgidriver*, *settextstyle*

restorecrtmode graphics.h

Funktion Stellt den Videomodus, der vor dem Aufruf von *initgraph* aktiv war, wieder her.

Syntax

```
void far restorecrtmode(void);
```

Beschreibung *initgraph* hält beim Start des Grafikpakets den ursprünglich aktiven Videomodus fest; mit *restorecrtmode* und *setgraphmode* können Sie deshalb zwischen Text- und Grafikmodus wechseln. *textmode* sollte für diesen Zweck nicht verwendet werden, diese Funktion dient nur zum Umschalten zwischen verschiedenen Textmodi.

Vor dem Aufruf von *restorecrtmode* muß das Grafikpaket mit *initgraph* aktiviert worden sein – ansonsten ist der »vorherige« Videomodus undefiniert.

Rückgabewert Keiner.

Siehe auch *getgraphmode*, *initgraph*, *setgraphmode*

sector graphics.h

Funktion Zeichnet ein ausgefülltes elliptisches »Kuchenstück«.

Syntax

```
void far sector(int x, int y, int stangle, int endangle, int xradius, int
yradius);
```

Beschreibung *sector* zeichnet ein elliptisches »Kuchenstück«, dessen Mittelpunkt durch (*x*,*y*) angegeben ist. *xradius* und *yradius* legen den horizontalen bzw. vertikalen Radius fest. *stangle* legt den Startpunkt des mit *radius* gezogenen Ellipsenbogens fest, *endangle* sein Ende. Die Endpunkte des Ellipsenbogens werden mit dem Ellipsenmittelpunkt verbunden. Danach füllt *sector* das entstandene »Kuchenstück« mit dem durch *setfillpattern* bzw. *setfillstyle* festgelegten Muster und in der dort angegebenen Farbe.

Die Angabe beider Winkel erfolgt in Grad, wobei entgegen dem Uhrzeigersinn gezählt wird (0 Grad = horizontal rechts vom Mittelpunkt, 90 Grad = senkrecht über dem Mittelpunkt usw.)

Wenn während des Ausfüllens eines »Kuchenstücks« mit *sector* ein Fehler auftritt, liefert der nächste Aufruf von *graphresult* den Wert -6 (grNoScanMem).

Rückgabewert Keiner.

Siehe auch *arc*, *circle*, *ellipse*, *getarccoords*, *getaspectratio*, *graphresult*, *pieslice*, *setfillpattern*, *setfillstyle*, *setgraphbufsize*

setactivepage graphics.h

Funktion Legt die aktive Grafikspeicherseite fest.

Syntax

```
void far setactivepage(int page);
```

Beschreibung *setactivepage* legt eine Grafikspeicherseite für nachfolgende Zeichenaktionen durch die Berechnung einer Basisadresse fest.

Mehrere Grafikspeicherseiten werden nur von EGA- und VGA-Karten sowie von der Hercules-Karte unterstützt. Mit Video-Adaptern dieser Typen ist es möglich, dem Benutzer eine Grafik-Seite zu zeigen, während »im Hintergrund« auf eine andere Seite gezeichnet wird.

Rückgabewert Keiner.

Siehe auch *setvisualpage*

setallpalette graphics.h

Funktion Setzt sämtliche Einträge einer Farbpalette.

Syntax

```
void far setallpalette(struct palettetype far *palette);
```

Beschreibung *setallpalette* setzt mit der von *palette* übergebenen Struktur des Typs *palettetype* sämtliche Einträge einer Farbpalette, wobei *size* hier festlegt, wieviele Einträge (ab dem ersten, also der »Farbe 0«) gesetzt werden sollen.

Jedes Element des Array *colors* ersetzt einen Eintrag der Farbpalette und kann einen Wert von -1 bis 63 haben. Wie bei dem entsprechenden BIOS-Aufruf bedeutet der »Farbwert« -1, daß der korrespondierende Eintrag der Farbpalette unverändert bleiben soll.

Mit *setallpalette* können alle oder einzelne Farben der EGA-/ VGA-Palette verändert werden. Änderungen wirken sich sofort am Bildschirm aus: wird ein Paletteneintrag geändert, ändern alle Elemente mit diesem Farbwert ihre Farbe.

Die MAXCOLORS-Konstante und die Struktur des Typs *palettetype* sind in graphics.h folgendermaßen definiert:

```
#define MAXCOLORS 15

struct palettetype {
   unsigned char size;

   signed char colors[MAXCOLORS+1];
};
```

size enthält die Anzahl der tatsächlich benutzten Einträge des Array *colors*, sein Wert ist vom aktiven Grafikmodus und dem verwendeten Treiber abhängig (CGA: 4 Farben, EGA: 16 Farben usw). Die Elemente des Array *colors* (0 bis *size*-1) enthalten die in der aktiven Palette gespeicherten Bitmuster (für CGA: Werte von 0 bis 3, für EGA und VGA: Werte von 0 bis 63).

Die folgenden symbolischen Konstanten sind in graphics.h für Farb-Bitmuster definiert:

Tabelle 4.6: Farbtabelle

CGA-Modus		EGA/VGA		
Name	**Nr.**	**Name**	**Nr.**	**Farbe**
BLACK	0	EGA_BLACK	0	schwarz
BLUE	1	EGA_BLUE	1	blau
GREEN	2	EGA_GREEN	2	grün
CYAN	3	EGA_CYAN	3	türkis
RED	4	EGA_RED	4	rot
MAGENTA	5	EGA_MAGENTA	5	violett
BROWN	6	EGA_BROWN	20	braun
LIGHTGRAY	7	EGA_LIGHTGRAY	7	hellgrau
DARKGRAY	8	EGA_DARKGRAY	56	dunkelgrau
LIGHTBLUE	9	EGA_LIGHTBLUE	57	hellblau
LIGHTGREEN	10	EGA_LIGHTGREEN	58	hellgrün
LIGHTCYAN	11	EGA_LIGHTCYAN	59	helles Türkis
LIGHTRED	12	EGA_LIGHTRED	60	hellrot
LIGHTMAGENTA	13	EGA_LIGHTMAGENTA	61	helles Violett
YELLOW	14	EGA_YELLOW	62	gelb
WHITE	15	EGA_WHITE	63	weiß

Änderungen einer Palettenfarbe wirken sich sofort am Bildschirm aus und betreffen jedes Vorkommen dieser Farbe.

setallpalette kann nicht zusammen mit dem IBM-8514-Treiber verwendet werden.

Rückgabewert

Wenn *setallpalette* mit ungültigen Argumenten aufgerufen wird, bleibt die aktuelle Farbpalette unverändert; ein folgender Aufruf von *graphresult* liefert in diesem Fall den Wert -11 (*grError*).

Siehe auch

getpalette, *getpalettesize*, *graphresult*, *setbkcolor*, *setcolor*, *setpalette*

setaspectratio graphics.h

Funktion

Verändert den Korrekturfaktor für das Höhen-/Seitenverhältnis des Bildschirms.

Syntax

```
void far setaspectratio(int xasp, int yasp);
```

Beschreibung Jeder Grafiktreiber (.BGI) enthält einen Wert zur Korrektur des physikalischen Höhen-/Seitenverhältnisses des Bildschirms, der automatisch von *arc*, *circle* und *pieslice* verwendet wird, um auch optisch runde Kreise zu erhalten.

Mit *setaspectratio* lassen sich die von den Zeichenfunktionen verwendeten Korrekturfaktoren direkt setzen (z.B. für spezielle Video-Adapter); die Angabe eines neuen Faktors erfolgt analog zu den von *getaspectratio* gelieferten Ergebnissen (*yasp* = 10000, *xasp* im entsprechenden Verhältnis dazu).

Rückgabewert Keiner.

Siehe auch *circle*, *getaspectratio*

setbkcolor graphics.h

Funktion Setzt die Hintergrundfarbe des Bildschirms.

Syntax

```
void far setbkcolor(int color);
```

Beschreibung *setbkcolor* setzt den Inhalt des ersten Eintrags der aktuellen Farbpalette auf den Wert von *color*. Für den Parameter *color* kann eine Nummer oder ein Name verwendet werden; hierfür sind die folgenden Konstanten in graphics.h als **enum** definiert:

Diese symbolischen Konstanten sind in graphics.h definiert.

Nr.	Name	Farbe	Nr.	Name	Farbe
0	BLACK	schwarz	8	DARKGRAY	dunkelgrau
1	BLUE	blau	9	LIGHTBLUE	hellblau
2	GREEN	grün	10	LIGHTGREEN	hellgrün
3	CYAN	türkis	11	LIGHTCYAN	helles Türkis
4	RED	rot	12	LIGHTRED	hellrot
5	MAGENTA	violett	13	LIGHTMAGENTA	helles Violett
6	BROWN	braun	14	YELLOW	gelb
7	LIGHTGRAY	hellgrau	15	WHITE	weiß

Folgende Aufrufe setzen Blau als Hintergrundfarbe:

```
setbkcolor (BLUE) /* or */ setbkcolor (1)
```

Auf CGA- und EGA-Karten ändert *setbkcolor* die Hintergrundfarbe durch Änderung des ersten Eintrags in der Palette.

Wenn bei einem EGA-/VGA-System die Paletteneinträge mit *setpalette* oder *setallpalette* geändert werden, kann die Verwendung der symbolischen Namen zu falschen Farben führen. Der Grund hierfür ist, daß *setbkcolor* sich immer auf die jeweilige Nummer in der Palette bezieht

und nicht auf eine bestimmte Farbe – außer der übergebene Parameter ist 0, wodurch die Hintergrundfarbe stets auf schwarz gesetzt wird.

Rückgabewert Keiner.

Siehe auch *getbkcolor*, *setallpalette*, *setcolor*, *setpalette*

setcolor graphics.h

Funktion Setzt die Farbe für spätere Zeichenaktionen.

Syntax

```
void far setcolor(int color);
```

Beschreibung *setcolor* legt fest, welcher Paletteneintrag zukünftig als Zeichenfarbe verwendet werden soll. *color* kann dabei Werte von 0 bis *getmaxcolor* haben.

Die Zeichenfarbe spezifiziert den Wert, auf den die Pixel beim Zeichnen von Linien etc. gesetzt werden. Die folgende Tabelle zeigt die für CGA bzw. EGA definierten Konstanten für die Zeichenfarbe (Nr. bezeichnet die Nummer der Palette).

	Konstante, die der Farbnummer zugewiesen wird (Pixelwert)		
Nr.	**1**	**2**	**3**
0	CGA_LIGHTGREEN	CGA_LIGHTRED	CGA_YELLOW
1	CGA_LIGHTCYAN	CGA_LIGHTMAGENTA	CGA_WHITE
2	CGA_GREEN	CGA_RED	CGA_BROWN
3	CGA_CYAN	CGA_MAGENTA	CGA_LIGHTGREY

Nr.	Name	Farbe	Nr.	Name	Farbe
0	BLACK	schwarz	8	DARKGRAY	dunkelgrau
1	BLUE	blau	9	LIGHTBLUE	hellblau
2	GREEN	grün	10	LIGHTGREEN	hellgrün
3	CYAN	türkis	11	LIGHTCYAN	helles Türkis
4	RED	rot	12	LIGHTRED	hellrot
5	MAGENTA	violett	13	LIGHTMAGENTA	helles Violett
6	BROWN	braun	14	YELLOW	gelb
7	LIGHTGRAY	hellgrau	15	WHITE	weiß

Zur Auswahl der Zeichenfarbe wird entweder die Farbnummer oder der entsprechende symbolische Name an *setcolor* übergeben.

Ein Beispiel: Im Modus CGAC0 enthält die Farbpalette vier Einträge – die Hintergrundfarbe BLACK(0) sowie LIGHTGREEN(1), LIGHTRED(2) und YELLOW(3). Durch den Aufruf *setcolor(3)* bzw. *setcolor(CGA_YELLOW)* wird als Zeichenfarbe YELLOW festgelegt.

Rückgabewert Keiner

Siehe auch *getcolor*, *getmaxcolor*, *graphresult*, *setallpalette*, *setbkcolor*, *setpalette*

setfillpattern graphics.h

Funktion Ermöglicht die freie Definition von Mustern für Flächenfüllungen.

Syntax

```
void far setfillpattern(char far *upattern, int color);
```

Beschreibung *setfillpattern* legt das Muster und die Farbe für nachfolgende Aufrufe von *fillellipse*, *fillpoly*, *floodfill*, *bar*, *bar3d*, *sector* und *pieslice* fest. Die Funktion arbeitet wie *setfillstyle*, verwendet aber ein benutzerdefiniertes 8*8-Muster anstatt eines vordefinierten Musters.

upattern ist ein Zeiger auf eine Folge von 8 Bytes (normalerweise ein Array), bei dem jedes Bit für ein Pixel in der durch *color* festgelegten Farbe steht. Jedes Byte legt acht nebeneinanderliegende Punkte einer Bildschirmzeile fest, die Bytes selbst sind »untereinander« angeordnet. Wird ein Bit auf 1 gesetzt, wird das zugehörige Pixel mit der Farbe gefüllt.

Rückgabewert Keiner.

Siehe auch *getfillpattern*, *getfillsettings*, *graphresult*, *sector*, *setfillstyle*

setfillstyle graphics.h

Funktion Setzt ein vordefiniertes Muster und die Farbe zur Flächenfüllung.

Syntax

```
void far setfillstyle(int pattern, int color);
```

Beschreibung *setfillstyle* setzt das aktuelle Muster und die dabei verwendete Farbe für Flächenfüllungen. Um ein durch den Programmierer definiertes Füllmuster (USER_FILL, Wert 12) zu setzen, muß *setfillpattern* verwendet werden.

Für das Füllmuster (*pattern*) definiert graphics.h mit dem Aufzählungstyp *fill_pattern* die folgenden Bitmuster:

Name	Wert	Beschreibung
EMPTY_FILL	0	Füllen mit der Hintergrundfarbe
SOLID_FILL	1	Füllen mit der Farbe *color*
LINE_FILL	2	----------
LTSLASH_FILL	3	///
SLASH_FILL	4	/// mit dicken Linien
BKSLASH_FILL	5	\\\ mit dicken Linien
LTBKSLASH_FILL	6	\\\
HATCH_FILL	7	leicht schraffiert

XHATCH_FILL	8	stark schraffiert, überkreuzend
INTERLEAVE_FILL	9	gestrichelte Linien
WIDE_DOT_FILL	10	Punkte, weit auseinander
CLOSE_DOT_FILL	11	Punkte, dicht nebeneinander
USER_FILL	12	durch den Programmierer definiert

setfillstyle erwartet einen dieser Werte als *pattern* und setzt das entsprechende Bitmuster.

Alle Werte außer *EMPTY_FILL* verwenden die aktuelle Füllfarbe für die Flächenfüllung. *EMPTY_FILL* verwendet die aktuelle Hintergrundfarbe.

Ein Aufruf von *setfillstyle* mit einem ungültigen Wert für *pattern* wird als Fehler gewertet: das aktuelle Füllmuster bleibt in diesen Fällen unverändert, ein nachfolgender Aufruf von *graphresult* liefert den Wert -11 (*grError*).

Rückgabewert Keiner.

Siehe auch *bar*, *bar3d*, *fillpoly*, *floodfill*, *getfillsettings*, *graphresult*, *piesclice*, *sector*, *setfillpattern*

setgraphbufsize — graphics.h

Funktion Legt die Größe des internen Grafikpuffers fest.

Syntax

```
unsigned far setgraphbufsize(unsigned bufsize);
```

Beschreibung Manche Grafikroutinen, wie die zur Flächenfüllung und zum Zeichnen von Polygonen, verwenden einen internen Puffer, der von *initgraph* beim Start des Grafikpakets auf dem Heap belegt und von *closegraph* wieder freigegeben wird.

Standardmäßig umfaßt dieser Puffer 4096 Bytes – über einen Aufruf von *setgraphbufsize* vor dem Start des Grafikpakets läßt er sich beliebig verkleinern (auf minimal 0 Bytes) oder vergrößern. *setgraphbufsize* teilt *initgraph* mit, wieviel Speicher für diesen internen Buffer bereitzustellen ist, wenn es *_graphgetmem* aufruft. Eine Vergrößerung ist angebracht, wenn beispielsweise ein Aufruf von *floodfill* den Fehler -7 erzeugt:
`Out of flood memory.`

setgraphbufsize muß vor *initgraph* aufgerufen werden. Nachdem *initgraph* aufgerufen wurde, werden alle weiteren *setgraphbufsize*-Aufrufe bis zum nächsten Aufruf von *closegraph* ignoriert.

Rückgabewert *setgraphbufsize* liefert die vorherige Größe des Grafikpuffers zurück.

Siehe auch *closegraph*, *_graphfreemem*, *_graphgetmem*, *initgraph*, *sector*

setgraphmode graphics.h

Funktion Schaltet in einen Grafikmodus um und löscht den Bildschirm.

Syntax

```
void far setgraphmode(int mode);
```

Beschreibung *setgraphmode* erwartet für *mode* einen für den benutzten Grafiktreiber gültigen Wert, löscht den Bildschirm, aktiviert den entsprechenden Grafikmodus und setzt alle Parameter des Grafikpakets (aktuelle Position, Palette, Farbe usw.) auf ihre Standardvorgaben zurück. Diese Funktion kann zusammen mit der Funktion *restorecrtmode* zum Umschalten zwischen Text- und Grafikmodi benutzt werden.

Rückgabewert Keiner. Wenn *setgraphmode* mit einem Wert für *mode* aufgerufen wird, den der aktuelle Grafiktreiber nicht unterstützt, bleibt der Videomodus unverändert, ein folgender Aufruf von *graphresult* liefert -10 (*grInvalidMode*).

Siehe auch *getgraphmode*, *getmoderange*, *graphresult*, *initgraph*, *restorecrtmode*

setlinestyle graphics.h

Funktion Setzt die Linienart und -dicke für spätere Zeichenaktionen.

Syntax

```
void far setlinestyle(int linestyle, unsigned upattern, int thickness);
```

Beschreibung *setlinestyle* setzt die Linienart für folgende Zeichenoperationen mit *line*, *lineto*, *rectangle*, *drawpoly*, *arc*, *circle*, *ellipse*, *pieslice* usw.

Die Struktur vom Typ *linesettingstype* ist in graphics.h folgendermaßen definiert:

```
struct linesettingstype {
   int linestyle;
   unsigned upattern;
   int thickness;
};
```

Für die Linienart (*linestyle*) definiert graphics.h mit dem Aufzählungstyp *line_styles* folgende Operatoren:

Name	Wert	Beschreibung
SOLID_LINE	0	durchgezogen
DOTTED_LINE	1	gepunktet
CENTER_LINE	2	Punkt Strich Punkt Strich...
DASHED_LINE	3	gestrichelt
USERBIT_LINE	4	benutzerdefiniert

Für die Dicke einer Linie (*thickness*) sind nur zwei Werte möglich, die ebenfalls in graphics.h definiert sind:

Name	Wert	Beschreibung
NORM_WIDTH	1	normale Dicke (1 Pixel)
THICK_WIDTH	3	»dicke« Dicke (3 Pixel)

Der Parameter *upattern* hat nur dann eine Wirkung, wenn *linestyle* den Wert USERBIT_LINE hat – in diesem Fall legen die einzelnen Bits das Punktmuster der Linie fest. Beispielsweise ergibt 0xFFFF für *upattern* eine durchgezogene Linie, 0x3333 oder 0x0F0F eine gestrichelte Linie usw.

Der Parameter *linestyle* wirkt sich nicht auf Kreisbögen, Kreise, Ellipsen, Kreis- oder Ellipsensektoren aus. Nur der Parameter *thickness* wird hierfür verwendet.

Wenn *setlinestyle* einen Wert von *upattern* für *linestyle != USERBIT_LINE* hat (d.h. ungleich 4 ist), wird der Parameter zwar ignoriert, muß aber trotzdem immer angegeben werden.

Rückgabewert

Keiner. Aufrufe von *setlinestyle* mit einem Wert für *linestyle* außerhalb des gültigen Bereichs werden als Fehler gewertet: In diesen Fällen bleibt die aktuelle Linienart unverändert, ein nachfolgender Aufruf von *graphresult* liefert das Ergebnis -11.

Siehe auch

arc, *bar3d*, *circle*, *drawpoly*, *ellipse*, *getlinesettings*, *graphresult*, *line*, *linerel*, *lineto*, *pieslice*, *rectangle*

setpalette graphics.h

Funktion

Ändert einen Eintrag der aktuellen Farbpalette.

Syntax

```
void far setpalette(int colornum, int color);
```

Beschreibung

setpalette setzt einen einzelnen (durch *colornum* bezeichneten) Eintrag der Farbpalette auf die mit *color* bezeichnete Farbe. Beispielsweise ändert *setpalette (0,5)* den ersten Eintrag in der aktuellen Palette (die Hintergrundfarbe) auf die Farbe mit der Nummer 5. Gibt es in der aktuellen Palette *size* Einträge, so darf *colornum* im Bereich zwischen 0 und *size -1* liegen.

Mit *setpalette* können einzelne (oder alle) Farben der EGA-/VGA-Palette verändert werden. Bei Verwendung einer CGA-Karte kann lediglich der erste Paletteneintrag (d.h. die Hintergrundfarbe) mit einem Aufruf von *setpalette* geändert werden.

Die folgenden symbolischen Farbkonstanten sind in der Header-Datei graphics.h definiert:

CGA-Modus Name	Wert	EGA/VGA Name	Wert	Farbe
BLACK	0	EGA_BLACK	0	schwarz
BLUE	1	EGA_BLUE	1	blau
GREEN	2	EGA_GREEN	2	grün
CYAN	3	EGA_CYAN	3	türkis
RED	4	EGA_RED	4	rot
MAGENTA	5	EGA_MAGENTA	5	violett
BROWN	6	EGA_BROWN	20	braun
LIGHTGRAY	7	EGA_LIGHTGRAY	7	hellgrau
DARKGRAY	8	EGA_DARKGRAY	56	dunkelgrau
LIGHTBLUE	9	EGA_LIGHTBLUE	57	hellblau
LIGHTGREEN	10	EGA_LIGHTGREEN	58	hellgrün
LIGHTCYAN	11	EGA_LIGHTCYAN	59	helles Türkis
LIGHTRED	12	EGA_LIGHTRED	60	hellrot
LIGHTMAGENTA	13	EGA_LIGHTMAGENTA	61	helles Violett
YELLOW	14	EGA_YELLOW	62	gelb
WHITE	15	EGA_WHITE	63	weiß

Die gültigen Farben sind abhängig vom aktuellen Grafiktreiber und Grafikmodus.

Änderungen wirken sich sofort am Bildschirm aus: wird ein Paletteneintrag geändert, ändern alle Elemente mit diesem Farbwert ihre Farbe.

setpalette kann nicht zusammen mit dem IBM 8514-Adapter verwendet werden. Verwenden Sie stattdessen *setrgbpalette*.

Rückgabewert Keiner. Wenn *setpalette* mit ungültigen Parametern aufgerufen wird, bleibt die aktuelle Farbpalette unverändert; ein folgender Aufruf von *graphresult* liefert in diesem Fall den Wert -11.

Siehe auch *getpalette*, *graphresult*, *setallpalette*, *setbkcolor*, *setcolor*, *setrgbpalette*

setrgbpalette graphics.h

Funktion Ändert einen Eintrag der aktuellen Farbpalette für den IBM-Adapter 8514.

Syntax

```
void far setrgbpalette(int colornum, int red, int green, int blue);
```

Beschreibung *setrgbpalette* kann für VGA-Treiber und den IBM 8514 verwendet werden. Für den IBM 8514-Adapter und für die VGA-Treiber im Modus mit 256 Farben liegt der gültige Bereich von *colornum* zwischen 0 und 255, für die übrigen VGA-Modi liegt er zwischen 0 und 15. Bei den Parametern *red*, *green* und *blue* werden nur das niederwertige Byte und bei diesem nur die sechs höchstwertigen Bits interpretiert.

colornum legt die Nummer des Farbeintrags fest, dessen Wert sich aus den Komponenten *red*, *green* und *blue* zusammensetzt.

Der Treiber IBM8514.BGI initialisiert die ersten 16 Paletteneinträge des IBM 8514-Adapters mit Werten, die den Standardfarben einer EGA- bzw. VGA-Karte entsprechen, und sorgt so dafür, daß »normale« Programme auch mit diesem Adapter lauffähig sind.

Rückgabewert Keiner.

Siehe auch *setpalette*

settextjustify graphics.h

Funktion Legt die Ausrichtung von Textausgaben durch Grafikroutinen fest.

Syntax

```
void far settextjustify(int horiz, int vert);
```

Beschreibung Die durch *settextjustify* gesetzten Parameter bestimmen, in welchem Verhältnis Textausgaben zur aktuellen Position (CP) horizontal und vertikal ausgerichtet werden.

Gemäß Vorgabe werden die Parameter LEFT_TEXT (horizontale Ausrichtung) und TOP_TEXT (vertikale Ausrichtung) gesetzt.

Für die Ausrichtung von Texten definiert der Aufzählungstyp *text_just* in graphics.h die folgenden Konstanten:

	Name	Wert	Beschreibung
horizontal	LEFT_TEXT	0	linksbündig
	CENTER_TEXT	1	zentriert
	RIGHT_TEXT	2	rechtsbündig
vertikal	BOTTOM_TEXT	0	Ausrichtung an der Grundlinie
	CENTER_TEXT	1	zentriert
	TOP_TEXT	2	Ausrichtung am oberen Rand

Wenn *horiz* auf LEFT_TEXT und *direction* auf HORIZ_DIR gesetzt wurde, dann wird nach einem Aufruf von *outtext(string)* die aktuelle Position um *textwidth(string)* nach rechts verschoben.

settextjustify kann nicht zusammen mit Stream-Funktionen oder im Textmodus verwendet werden.

Rückgabewert Keiner. Falls ein ungültiger Wert an *settextjustify* übergeben wurde, liefert *graphresult* -11 zurück; die Ausrichtung des Texts bleibt dann unverändert.

Siehe auch *gettextsettings*, *graphresult*, *outtext*, *settextstyle*

settextstyle graphics.h

Funktion

Legt den Font, die Schreibrichtung und die Größe folgender Textausgaben im Grafikmodus fest.

Syntax

```
void far settextstyle(int font, int direction, int charsize);
```

Beschreibung

settextstyle setzt einen Font (*font*), die Schreibrichtung (*direction*) und den Vergrößerungsfaktor (*charsize*). Ein Aufruf von *settextstyle* beeinflußt alle nachfolgenden Textausgaben mit *outtext* bzw. *outtextxy*.

font: Ein 8*8-Bitmap-Font und verschiedene Vektor-Fonts stehen zur Verfügung. Der Aufzählungstyp *font_names* definiert die folgenden Konstanten in graphics.h:

Name	Wert	Beschreibung
DEFAULT_FONT	0	8 * 8 Bitmap-Font (Standard)
TRIPLEX_FONT	1	Vektor-Font »Triplex«
SMALL_FONT	2	Kleiner Vektor-Font
SANSSERIF_FONT	3	Vektor-Font »Sans serif«
GOTHIC_FONT	4	Vektor-Font »Gothic«
SCRIPT_FONT	5	Vektor-Font »Script«
SIMPLEX_FONT	6	Vektor-Font »Simplex«
TRIPLEX_SCR_FONT	7	Vektor-Font »Triplex Script«
COMPLEX_FONT	8	Vektor-Font »Complex«
EUROPEAN_FONT	9	Vektor-Font »European«
BOLD_FONT	10	Vektor-Font »Bold«

Der vordefinierte Bitmap-Font (DEFAULT_FONT) ist Teil des Grafikpakets und immer verfügbar. Die anderen Fonts sind in separaten Dateien (*.CHR) gespeichert und können mit *settextstyle* entweder nur bei Bedarf einzeln von der Diskette geladen werden oder mit dem Zusatzprogramm BGIOBJ in .OBJ-Dateien konvertiert und mit *registerbgifont* »registriert« und so in das Programm gelinkt werden.

Der Wert von *direction* gibt an, ob Textausgaben von links nach rechts (d.h. horizontal) oder von unten nach oben (d.h. in vertikaler Richtung) erfolgen. Mögliche Werte für *direction* sind:

Name	Wert	Beschreibung
HORIZ_DIR	0	von links nach rechts (Standard)
VERT_DIR	1	Ausgabe von unten nach oben

charsize legt den Vergrößerungsfaktor (0 bis 10) für Textausgaben fest. Falls *charsize* ungleich Null ist, können sowohl Zeichen aus Vektor- als

auch aus Bitmap-Fonts davon betroffen sein. Der Wert Null für *charsize* ist nur auf Vektor-Fonts anwendbar.

- Falls *charsize* gleich 1 ist, zeigen *outtext* und *outtextxy* Zeichen des 8*8 Bitmap-Fonts in einem Rechteck mit 8*8 Pixeln am Bildschirm an.
- Falls *charsize* gleich 2 ist, zeigen diese Ausgabefunktionen Zeichen des 8*8 Bitmap-Fonts in einem Rechteck mit 16*16 Pixeln (usw., bis zu einem Limit vom Zehnfachen der normalen Größe) an.
- Falls *charsize* gleich 0 ist, vergrößern die Ausgabefunktionen *outtext* und *outtextxy* den Text bei Verwendung eines Vektor-Fonts mit dem Standardvergrößerungsfaktor (4) bzw. mit dem vom Benutzer mit *setusercharsize* definierten Vergrößerungsfaktor.

Mit *textheight* und *textwidth* können Sie jederzeit die tatsächlichen Abmessungen des Textes abrufen.

Rückgabewert Keiner.

Siehe auch *gettextsettings*, *graphresult*, *installuserfont*, *settextjustify*, *setusercharsize*, *textheight*, *textwidth*

setusercharsize graphics.h

Funktion Legt voneinander unabhängige Vergrößerungsfaktoren in X- und Y-Richtung für Vektor-Fonts fest.

Syntax

```
void far setusercharsize(int multx, int divx, int multy, int divy);
```

Beschreibung *setusercharsize* erlaubt im Gegensatz zu *settextstyle* für Vektor-Fonts voneinander unabhängige Vergrößerungsfaktoren in X- und Y-Richtung: die Breite eines Zeichens wird zuerst mit *multx* multipliziert und danach durch *divx* dividiert. Dasselbe gilt für die Zeichenhöhe und die Parameter *multy* bzw. *divy*.

Soll beispielsweise Text doppelt so breit und 50 Prozent höher als in der Standardvorgabe festgelegt ausgegeben werden, muß *setusercharsize* mit den Parametern

```
multx = 2;   divx = 1;
multy = 3;   divy = 2;
```

aufgerufen werden.

Die von *setusercharsize* gesetzten Werte sind nur dann aktiv, wenn *charsize* durch einen vorausgehenden Aufruf von *settextstyle* auf 0 gesetzt wurde.

Rückgabewert Keiner.

Siehe auch *gettextsettings*, *graphresult*, *settextstyle*

setviewport graphics.h

Funktion Setzt ein Zeichenfenster.

Syntax

```
void far setviewport(int left, int top, int right, int bottom, int clip);
```

Beschreibung *setviewport* legt mit (*left*,*top*) die obere linke Ecke und mit (*right*, *bottom*) die untere rechte Ecke eines neuen Zeichenfensters fest und arbeitet mit absoluten Koordinaten. Der Aufruf

```
setviewport(0,0,100,100,1);
```

setzt ein Zeichenfenster von 100 Pixeln Breite, das sich in der oberen linken Ecke des Bildschirms befindet – unabhängig davon, ob zuvor ein anderes Zeichenfenster gesetzt war oder nicht.

setviewport setzt die aktuelle Position (CP) auf den Punkt (0,0) innerhalb des neuen Fensters. Der »Fensterinhalt« wird durch diese Funktion nicht verändert.

Der Parameter *clip* legt fest, ob Zeichenaktionen am Rand des Fensters abgeschnitten werden sollen (*clip* = 1) oder nicht (*clip* = 0).

Rückgabewert Keiner. Der Aufruf von *setviewport* mit Werten, die die Grenzen des Bildschirms überschreiten, läßt die aktuellen Grafikeinstellungen unverändert. Ein nachfolgender Aufruf von *graphresult* liefert in diesem Fall -11.

Siehe auch *clearviewport*, *getviewsettings*, *graphresult*

setvisualpage graphics.h

Funktion Wählt eine von mehreren Grafikspeicherseiten zur Anzeige aus.

Syntax

```
void far setvisualpage(int page);
```

Beschreibung *setvisualpage* wählt die Grafikseite *page* zur Anzeige auf dem Bildschirm aus.

Rückgabewert Keiner.

Siehe auch *graphresult*, *setactivepage*

setwritemode graphics.h

Funktion Legt den Schreibmodus für Zeichenoperationen von Linien im Grafikmodus fest.

Syntax

```
void far setwritemode(int mode);
```

Beschreibung Die beiden möglichen Werte für den Parameter *mode* sind in graphics.h als Konstanten definiert:

```
COPY_PUT = 0 /* MOV */
XOR_PUT = 1  /* XOR */
```

setwritemode legt fest, ob Zeichenaktionen mit *line*, *linerel*, *lineto*, *rectangle* und *drawpoly* den vorherigen Bildschirminhalt direkt überschreiben (COPY_PUT) oder ob dazu eine XOR-Operation verwendet wird. COPY_PUT verwendet die Assembler-Anweisung **MOV** und überschreibt daher den vorherigen Bildschirminhalt. Mit XOR_PUT läßt sich die gezeichnete Linie durch einen weiteren Zeichenbefehl im selben Modus ohne Zerstörung des Hintergrundes wieder vom Bildschirm entfernen.

Rückgabewert Keiner.

Siehe auch *drawpoly, line, linerel, lineto, putimage*

textheight graphics.h

Funktion Liefert die Höhe eines Strings in Pixel zurück.

Syntax

```
int far textheight(char far *textstring);
```

Beschreibung *textheight* berechnet aus *textstring* sowie dem aktuellen Font und dem Vergrößerungsfaktor den vertikalen Platzbedarf einer Textzeile.

Diese Funktion ist sehr hilfreich zur Berechnung von Zeichenabständen, der Höhe eines Zeichenfensters, der Größe von Texten, die in eine Grafik oder ein kleines Fenster eingepaßt werden sollen, usw.

Wenn beispielsweise der Standardfont (8 * 8 Pixel) zusammen mit einem Vergrößerungsfaktor von 1 (durch *settextstyle*) gesetzt ist, dann liefert *textheight* für den String *TurboC++* das Ergebnis 8 zurück.

Der Platzbedarf von Strings sollte *grundsätzlich* über *textheight* und *textwidth* ermittelt werden. Programme, die diese Funktionen anstelle von Konstanten verwenden, erfordern bei einem Wechsel der Schrift oder des Grafiktreibers keine Modifikationen.

Rückgabewert Die Funktion liefert die Texthöhe in Pixel zurück.

Siehe auch *gettextsettings, outtext, outtextxy, settextstyle, textwidth*

textwidth graphics.h

Funktion Liefert die Breite eines Strings in Pixel zurück.

Syntax

```
int far textwidth(char far *textstring);
```

Beschreibung

textwidth berechnet aus *textstring* sowie dem aktuellen Font und dem Vergrößerungsfaktor den horizontalen Platzbedarf einer Textzeile. Wenn beispielsweise der Standardfont (8 * 8 Pixel) zusammen mit dem Vergrößerungsfaktor 1 gesetzt ist, dann liefert *textwidth* für den String *Turbo* das Ergebnis 40 (= 5 * 8) zurück.

Der Platzbedarf von Strings sollte *grundsätzlich* über *textheight* und *textwidth* ermittelt werden, weil z.B. das Zentrieren von Titeln, Einpassen von Beschriftungen usw. auf diese Weise wesentlich einfacher geht. Programme, die diese Funktionen anstelle von Konstanten verwenden, erfordern bei einem Wechsel der Schrift oder des Grafiktreibers keine Modifikationen.

Rückgabewert

Die Funktion liefert die Textbreite in Pixel zurück.

Siehe auch

gettextsettings, *outtext*, *outtextxy*, *settextstyle*, *textheight*

K A P I T E L 5

DOS-Funktionen

Mit Ausnahme der Funktionen *brk* und *sbrk* (die sowohl unter DOS als auch unter UNIX verfügbar sind) sind die folgenden Funktionen nur für 16-Bit DOS-Anwendungen verfügbar. Im Kapitel 3 des *Referenzhandbuchs* finden Sie weitere Funktionen, die zum Teil auch in 16-Bit DOS-Anwendungen verwendet werden können. Die Beschreibungen solcher Funktionen, auf die im Abschnitt **Siehe auch** in diesem Kapitel verwiesen wird, finden Sie im Kapitel 3 des *Referenzhandbuchs*.

absread — dos.h

Funktion

Liest einzelne Diskettensektoren.

Syntax

```
int absread(int drive, int nsects, long lsect, void *buffer);
```

Beschreibung

Diese Funktion ignoriert die logische Struktur einer Diskette (d.h. die Unterteilung in FAT, Verzeichnisse, Dateien etc.) und bearbeitet einzelne Sektoren direkt.

absread verwendet den DOS-Interrupt 0x25 zum Lesen von Sektoren.

drive ist die Laufwerksnummer (A = 0, B = 1, C = 2, ...),
nsects die Anzahl der Sektoren, die gelesen werden sollen,
lsect die Nummer des ersten zu lesenden Sektors und
buffer der Zeiger auf einen Speicherbereich, in den die Daten gelesen werden sollen.

Mit einem Aufruf von *absread* können maximal 64 KByte gelesen werden. Ist der Puffer kleiner, bestimmt er die Obergrenze.

Rückgabewert

Bei fehlerfreier Ausführung liefert *absread* den Wert 0 zurück. Im Fall eines Fehlers wird die globale Variable *errno* auf den vom System gelieferten Wert des Registers AX gesetzt. *absread* hat dann den Rückgabewert -1.

Siehe auch

abswrite, *biosdisk*

abswrite — dos.h

Funktion Schreibt einzelne Diskettensektoren.

Syntax

```
int abswrite(int drive, int nsects, long lsect, void *buffer);
```

Beschreibung Diese Funktion ignoriert die logische Struktur einer Diskette (d.h. die Unterteilung in FAT, Verzeichnisse, Dateien etc.) und bearbeitet einzelne Sektoren direkt.

abswrite kann Dateien, Verzeichnisse und FATs überschreiben, wenn Sie die Parameter entsprechend angeben.

abswrite schreibt Sektoren über den DOS-Interrupt 0x26.

drive ist die Laufwerksnummer (A = 0, B = 1, C = 2, ...),
nsects die Anzahl der Sektoren, die geschrieben werden sollen,
lsect die Nummer des ersten zu schreibenden Sektors und
buffer der Zeiger auf einen Speicherbereich, dessen Inhalt geschrieben werden soll.

Mit einem Aufruf von *abswrite* können maximal 64 KByte geschrieben werden. Ist der Puffer kleiner, bestimmt er die Obergrenze.

Rückgabewert Bei fehlerfreier Ausführung liefert *abswrite* den Wert 0 zurück. Im Fall eines Fehlers wird die globale Variable *errno* auf den vom System gelieferten Wert des Registers AX gesetzt. *abswrite* hat dann den Rückgabewert -1.

Siehe auch *absread*, *biosdisk*

allocmem, _dos_allocmem — dos.h

Funktion Reserviert DOS-Speichersegmente für ein Programm.

Syntax

```
int allocmem(unsigned size, unsigned *segp);
unsigned _dos_allocmem(unsigned size, unsigned *segp);
```

Beschreibung *allocmem* und *_dos_allocmem* verwenden die DOS-Funktion 0x48, um einen (zusätzlichen) Speicherblock für das aktuelle Programm zu reservieren, und liefern die Segmentadresse des belegten Blocks zurück.

Über den Parameter *size* wird die Größe des zu belegenden Blocks in Paragraphen (d.h. Abschnitten zu jeweils 16 Byte) angegeben. *segp* zeigt auf das WORD, dem die Segmentadresse des belegten Blocks zugewiesen wurde.

Ist kein Block der angegebenen Größe verfügbar, führt *allocmem* keine Zuweisung an das WORD durch, auf das *segp* zeigt. *_dos_allocmem* speichert in diesem Fall die Größe (in Paragraphen) des größten verfügbaren Blocks in dem WORD, auf das *segp* zeigt.

Die durch *allocmem* und *_dos_allocmem* belegten Speicherbereiche sind paragraphenweise ausgerichtet.

allocmem und *_dos_allocmem* können nicht mit *malloc* koexistieren.

Rückgabewert

allocmem liefert den Wert -1, wenn ein Speicherbereich der gewünschten Größe reserviert werden konnte; ansonsten wird die Größe des größten noch verfügbaren Speicherblocks (in Paragraphen) zurückgeliefert.

_dos_allocmem liefert den Wert 0, wenn ein Speicherbereich der gewünschten Größe reserviert werden konnte. Andernfalls wird der DOS-Fehlercode zurückgeliefert und das WORD, auf das *segp* zeigt, auf die Größe des größten noch verfügbaren Speicherblocks (in Paragraphen) gesetzt.

Ist *allocmem* bzw. *_dos_allocmem* nicht erfolgreich, wird die globale Variable *_doserrno* gesetzt, und die globale Variable *errno* erhält den folgenden Wert:

ENOMEM Nicht genug Platz im Hauptspeicher

Siehe auch

coreleft, *freemem*, *malloc*, *setblock*

bioscom — bios.h

Funktion

Steuert Ein- und Ausgaben über serielle Schnittstellen.

Syntax

```
int bioscom(int cmd, char abyte, int port);
```

Beschreibung

Diese Funktion ermöglicht das Setzen von Parametern der seriellen Schnittstellen sowie die Ein- und Ausgabe von Daten über die in *port* angegebene Schnittstelle.

Der Parameter *port* legt fest, mit welcher der beiden Schnittstellen gearbeitet wird (0 = COM1, 1 = COM2, usw.).

Der Parameter *cmd* kann Werte von 0 bis 3 annehmen:

- 0 setzt die Übertragungsparameter auf den in *abyte* enthaltenen Wert;
- 1 gibt den Wert von *abyte* über die serielle Schnittstelle aus;
- 2 liest ein Zeichen über die serielle Schnittstelle;
- 3 liefert den aktuellen Status der seriellen Schnittstelle.

Der Parameter *abyte* ist eine Kombination von Bitwerten (dabei wird aus jeder Gruppe ein Wert ausgewählt):

Bitwert	Bedeutung
0x00	110 Baud
0x20	150 Baud
0x40	300 Baud
0x60	600 Baud
0x80	1200 Baud
0xA0	2400 Baud
0xC0	4800 Baud
0xE0	9600 Baud
0x00	keine Paritätsprüfung
0x08	ungerade Parität
0x18	gerade Parität
0x00	1 Stopbit
0x04	2 Stopbits
0x02	7 Datenbits
0x03	8 Datenbits

Die einzelnen Werte müssen miteinander verbunden werden. Ein Beispiel: die Parameter »9600 Baud, ungerade Parität, 1 Stopbit, 8 Datenbits« werden mit dem Wert 0xEB von *abyte* gesetzt, der sich aus (0xE0 | 0x08 | 0x00 | 0x03) ergibt.

bioscom verwendet den BIOS-Interrupt 0x14.

Rückgabewert

Das höherwertige Byte des von *bioscom* zurückgelieferten Integers enthält für alle Werte von *cmd* eine Statusinformation, die in einzelnen Bits kodiert ist. Das niederwertige Byte hängt vom *cmd*-Wert ab, die Bedeutung der höherwertigen Bits ist:

Bit 15	Time out
Bit 14	Schieberegister für Übertragung leer
Bit 13	Übertragungspuffer leer
Bit 12	Break-Signal erhalten
Bit 11	Framing-Fehler
Bit 10	Paritätsfehler
Bit 09	Überlauf
Bit 08	Daten bereit

Wenn der in *abyte* enthaltene Wert nicht gesendet werden konnte, wird Bit 15 auf 1 gesetzt, in allen anderen Fällen werden die übrigen höher- und niederwertigen Bits in geeigneter Weise gesetzt. Bei einem Framing-Fehler wird beispielsweise das Bit 11 auf 1 gesetzt.

Wenn *cmd* den Wert 2 hat und ein Zeichen fehlerfrei gelesen werden konnte, dann sind sämtliche Bits des höherwertigen Bytes auf »0« gesetzt

und das niederwertige Byte enthält das gelesene Zeichen. Im Fall eines Fehlers wird mindestens eines der höherwertigen Bits auf 1 gesetzt.

Für die Werte 0 und 3 von *cmd* liefert *bioscom* einen Wert zurück, dessen höherwertiges Byte sich aus den oben beschriebenen Bits zusammensetzt. Zusätzlich enthält das niederwertige Byte die folgenden Informationen:

Bit 07	Received line signal detect
Bit 06	Ring indicator
Bit 05	Data set ready
Bit 04	Clear to send
Bit 03	Change in receive line signal detector
Bit 02	Trailing edge ring detector
Bit 01	Change in data set ready
Bit 00	Change in clear to send

biosdisk bios.h

Funktion Direkter Zugriff auf Disketten/Festplatten über BIOS-Interrupt.

Syntax

```
int biosdisk(int cmd, int drive, int head, int track, int sector,
             int nsects, void *buffer);
```

Beschreibung Diese Funktion verwendet den BIOS-Interrupt 0x13 für direkte Zugriffe auf Disketten- und Festplattenlaufwerke.

Der Parameter *drive* legt fest, welches Laufwerk angesprochen werden soll, und wird wie folgt interpretiert:

- Werte im Bereich von 0x00 bis 0x7F sprechen Diskettenlaufwerke an – 0x00 steht für das erste Laufwerk, 0x01 für das zweite usw.
- Werte im Bereich von 0x80 bis 0xFF adressieren Festplattenlaufwerke – 0x80 steht für das erste Festplattenlaufwerk, 0x81 für das zweite usw.

Bei Festplatten wird nicht die Partition, sondern das physikalische Laufwerk angegeben. Falls die Festplatte partitioniert ist, muß die Interpretation der Partitionstabelle durch das aufrufende Programm erfolgen.

Der Parameter *cmd* legt die durchzuführende Operation fest. Ob die übrigen Parameter benötigt werden, hängt vom Wert des *cmd*-Parameters ab.

Für *cmd* sind auf allen Computern der IBM PC-Serie und Kompatiblen die folgenden Werte möglich:

0 führt ein »hartes« Reset des Controllers durch und ignoriert sämtliche anderen Parameter.

1 liefert den Status der zuletzt ausgeführten Operation zurück und ignoriert alle anderen Parameter.

2 liest einen oder mehrere Sektoren von der Diskette/Festplatte in den Speicher. Der erste zu lesende Sektor ist durch die Parameter *head*, *track* und *sector* festgelegt, die Anzahl der zu lesenden Sektoren durch *nsects*. Die Daten werden mit 512 Bytes pro Sektor in *buffer* eingelesen.

3 schreibt einen oder mehrere Sektoren aus dem Speicher auf die Diskette/Festplatte. Der erste zu schreibende Sektor ist durch die Parameter *head*, *track* und *sector* festgelegt, die Anzahl der zu schreibenden Sektoren durch *nsects*. Die Daten werden mit 512 Bytes pro Sektor aus *buffer* geschrieben.

4 prüft einen oder mehrere Sektoren. Der erste zu überprüfende Sektor ist durch die Parameter *head*, *track* und *sector* festgelegt, die Anzahl der zu überprüfenden Sektoren durch *nsects*.

5 formatiert die durch *track* und *head* angegebene Spur. *buffer* muß in diesem Fall auf eine Tabelle mit Parametern für Sektorzahl, -nummern, -länge usw. zeigen. Lesen Sie die Beschreibung dieser Tabelle und der Formatierung im *Technischen Referenzhandbuch* für den IBM PC nach.

Die folgenden Werte für *cmd* sind nur auf einem XT, AT, PS/2 oder Kompatiblen verwendbar:

6 formatiert die durch *track* und *head* angegebene Spur und trägt Kennzeichenbits für defekte Sektoren ein;

7 formatiert ab einer bestimmten Spur;

8 liefert die Parameter des aktuellen Laufwerks in den ersten vier Bytes von *buffer* zurück;

9 initialisiert die Parametertabellen für Festplatten;

10 long read – liest 512 Bytes plus 4 zusätzliche Prüfbytes pro Sektor;

11 long write – schreibt 512 Bytes plus 4 zusätzliche Prüfbytes pro Sektor;

12 seek – setzt den Schreib-/Lesekopf auf eine bestimmte Spur;

13 modifiziert den Reset;

14 liest den Sektorpuffer;

15 schreibt den Sektorpuffer;

16 prüft, ob das angegebene Laufwerk betriebsbereit ist;

17 kalibriert das Laufwerk neu;

18 Test des Controller-RAM;

19 Test des Laufwerks;

20 Test der internen Controller-Funktionen.

biosdisk arbeitet unterhalb der Dateiebene direkt mit den Sektoren. *Die Funktion kann Dateiinhalte und Verzeichnisse auf einer Festplatte zerstören.*

Rückgabewert

Sämtliche Aufrufe von *biosdisk* liefern ein Statusbyte zurück, für das die folgenden Werte definiert sind:

0x00	Fehlerfreie Ausführung
0x01	Unbekannter Befehl (ungültiger Wert von *cmd*)
0x02	Adreß-Kennzeichnung nicht gefunden
0x03	Schreibgeschützt (nur für Diskettenlaufwerke)
0x04	Sektor nicht gefunden/ungültige Sektornummer
0x05	RESET konnte nicht ausgeführt werden (Festplatte)
0x06	Diskettenwechsel seit letzter Operation
0x07	Parametertabelle ungültig
0x08	DMA-Fehler
0x09	Überschreitung der Segmentgrenze von 64 KByte bei DMA-Operation
0x0A	Sektor ist als »bad« markiert
0x0B	Spur ist als »bad« markiert
0x0C	Spur ungültig
0x10	Sektordaten ungültig (CRC/ECC-Prüfsummenfehler)
0x11	Datenfehler, durch CRC/ECC korrigiert (nur Festplatten)
0x20	Controller-Fehler
0x40	Suchen-Operation fehlgeschlagen
0x80	Laufwerk nicht bereit (time out)
0xAA	Platte nicht bereit (Festplattensysteme)
0xBB	Undefinierter Fehler (Festplattensysteme)
0xCC	Schreibfehler
0xEO	Statusfehler
0xFF	Abtastung fehlgeschlagen

Der Wert 0x11 zeigt an sich keinen Fehler an und ist diagnostischer Natur. Die gelesenen Daten sind korrekt bzw. wurden durch den Prüfsummenmechanismus korrigiert.

Siehe auch

absread, *abswrite*

_bios_disk bios.h

Funktion

Direkter Zugriff auf Disketten/Festplatten über BIOS-Interrupt.

Syntax

```
unsigned _bios_disk(unsigned cmd, struct diskinfo_t *dinfo);
```

Beschreibung

Diese Funktion verwendet den BIOS-Interrupt 0x13 für direkte Zugriffe auf Disketten- und Festplattenlaufwerke. Die Art der auszuführenden Operation wird über den Wert von *cmd* festgelegt. *dinfo* zeigt auf die Struktur *diskinfo_t* mit den weiteren Argumenten, die für die Operation benötigt werden.

Die *diskinfo_t*-Struktur ist in bios.h definiert und hat folgendes Format:

```
struct diskinfo_t {
    unsigned drive, head, track, sector, nsectors;
    void far *buffer;
};
```

Der Parameter *drive* legt fest, welches Laufwerk angesprochen werden soll, und wird wie folgt interpretiert:

- Werte im Bereich von 0x00 bis 0x7F sprechen Diskettenlaufwerke an – 0x00 steht für das erste Laufwerk, 0x01 für das zweite usw.
- Werte im Bereich von 0x80 bis 0xFF adressieren Festplattenlaufwerke – 0x80 steht für das erste Festplattenlaufwerk, 0x81 für das zweite usw.

Bei Festplatten wird nicht die Partition, sondern das physikalische Laufwerk angegeben. Falls die Festplatte partitioniert ist, muß die Interpretation der Partitionstabelle durch das aufrufende Programm erfolgen.

Ob die übrigen Parameter in der Struktur *diskinfo_t* benötigt werden, hängt vom Wert des *cmd*-Parameters ab.

Für *cmd* sind folgende Werte in bios.h definiert:

_DISK_RESET
: – führt ein »hartes« Reset des Controllers durch und ignoriert alle anderen Parameter von *diskinfo_t*.

_DISK_STATUS
: – liefert den Status der zuletzt ausgeführten Operation zurück und ignoriert alle anderen Parameter von *diskinfo_t*.

_DISK_READ
: – liest einen oder mehrere Sektoren von der Diskette/Festplatte in den Speicher. Der erste zu lesende Sektor ist durch die Parameter *head*, *track* und *sector* festgelegt, die Anzahl der zu lesenden Sektoren durch *nsectors*. Die Daten werden, 512 Bytes pro Sektor, in *buffer* eingelesen. Ist die Operation erfolgreich, ist das höherwertige Byte des zurückgelieferten Werts 0, und das nie-

derwertige Byte enthält die Anzahl der Sektoren. Bei einem Fehler hat das höherwertige Byte des Rückgabewerts einen der folgenden Werte:

0x01	Unbekannter Befehl
0x02	Adreßmarkierung nicht gefunden
0x03	Schreibgeschützt (nur für Diskettenlaufwerke)
0x04	Sektor nicht gefunden/ungültige Sektornummer
0x05	RESET konnte nicht ausgeführt werden (Festplatte)
0x06	Diskettenwechsel seit letzter Operation
0x07	Parametertabelle ungültig
0x08	DMA-Fehler
0x09	Überschreitung der Segmentgrenze von 64 KByte bei DMA-Operation
0x0A	Sektor ist als »bad« markiert
0x0B	Spur ist als »bad« markiert
0x0C	Spur ungültig
0x10	Sektordaten ungültig (CRC/ECC-Prüfsummenfehler)
0x11	Datenfehler, durch CRC/ECC korrigiert (nur Festplatten)
0x20	Controller-Fehler
0x40	Suchen-Operation fehlgeschlagen
0x80	Laufwerk nicht bereit (time out)
0xAA	Platte nicht bereit (Festplattensysteme)
0xBB	Undefinierter Fehler (Festplattensysteme)
0xCC	Schreibfehler
0xEO	Statusfehler
0xFF	Abtastung fehlgeschlagen

Der Wert 0x11 zeigt an sich keinen Fehler an und ist diagnostischer Natur. Die gelesenen Daten sind korrekt bzw. wurden durch den Prüfsummenmechanismus korrigiert.

_DISK_WRITE

– schreibt einen oder mehrere Sektoren aus dem Speicher auf die Diskette/Festplatte. Der erste zu schreibende Sektor ist durch die Parameter *head*, *track* und *sector* festgelegt, die Anzahl der zu schreibenden Sektoren durch *nsectorss*. Die Daten werden mit 512 Bytes pro Sektor aus *buffer* geschrieben. Die Rückgabewerte finden Sie in der obigen Beschreibung von *_DISK_READ*.

_DISK_VERIFY

– prüft einen oder mehrere Sektoren. Der erste zu überprüfende Sektor ist durch die Parameter *head*, *track* und *sector* festgelegt, die Anzahl der zu überprüfenden Sektoren durch *nsectors*. Die Rückgabewerte finden Sie in der obigen Beschreibung von *_DISK_READ*.

_DISK_FORMAT
– formatiert die durch *track* und *head* angegebene Spur. *buffer* muß in diesem Fall auf eine Tabelle mit Parametern für Sektorzahl, -nummern, -länge usw. zeigen. Lesen Sie die Beschreibung dieser Tabelle und der Formatierung im *Technischen Referenzhandbuch* für den IBM PC nach.

Rückgabewert *_bios_disk* liefert den durch den BIOS-Interrupt 0x13 gesetzten Wert im AX-Register zurück.

Siehe auch *absread*, *abswrite*, *biosdisk*

bioskey — bios.h

Funktion Tastaturabfrage über BIOS-Interrupt.

Syntax
```
int bioskey(int cmd);
```

Beschreibung Diese Funktion verwendet den BIOS-Interrupt 0x16 zur Kommunikation mit der Tastatur. Die Art der auszuführenden Operation wird über den Wert von *cmd* festgelegt.

Rückgabewert Der Rückgabewert hängt von der Operation ab, die über den Wert von *cmd* festgelegt wurde.

0 Falls die niederwertigen 8 Bit ungleich null sind, liefert *bioskey* das nächste ASCII-Zeichen aus dem Tastaturpuffer oder wartet auf die Eingabe eines Zeichens und liefert dieses Zeichen zurück. Bei der Eingabe eines »normalen« Zeichens liefert *bioskey* den entsprechenden ASCII-Code im niederwertigen Byte. Für Spezialzeichen wie Funktions- und Cursortasten hat das niederwertige Byte den Wert 0, das höherwertige Byte enthält den erweiterten Tastaturcode (siehe *Technisches Referenzhandbuch* für den IBM PC).

1 prüft, ob ein Zeichen anliegt. Wenn dies nicht der Fall ist, wird der Wert 0 zurückgeliefert, ansonsten der entsprechende Tastencode. Ein Rückgabewert von 0xFFFFF(-1) bedeutet, daß [*Strg*][*Untbr*] gedrückt wurde. Über diese Funktion gelesene Zeichen werden nicht »abgeholt« – sie verbleiben im Tastaturpuffer und werden nach dem nächsten Aufruf von *bioskey* mit dem *cmd*-Wert 0 ausgegeben.

2 liefert den aktuellen Zustand der Umschalttasten ([*Shift*], [*Alt*] usw.). Der Rückgabewert ergibt sich aus den über ODER-Operationen verbundenen Bitwerten:

Bit 7	0x80	[*Einfg*] aktiv
Bit 6	0x40	[*Shift*] aktiv
Bit 5	0x20	[*NumLock*] aktiv
Bit 4	0x10	[*Rollen*] aktiv
Bit 3	0x08	[*Alt*] gedrückt
Bit 2	0x04	[*Strg*] gedrückt
Bit 1	0x02	[*Shift*] links gedrückt
Bit 0	0x01	[*Shift*] rechts gedrückt

_bios_keybrd — bios.h

Funktion Tastaturabfrage über BIOS-Interrupt.

Syntax

```
unsigned _bios_keybrd(unsigned cmd);
```

Beschreibung Die Funktion *_bios_keybrd* verwendet den BIOS-Interrupt 0x16 zur Kommunikation mit der Tastatur. Die Art der auszuführenden Operation wird über den Wert von *cmd* festgelegt.

Rückgabewert Der Rückgabewert von *_bios_keybrd* hängt von der Operation ab, die über den Wert von *cmd* festgelegt wurde.

Folgende Werte sind für *cmd* in bios.h definiert:

_KEYBRD_READ
: – falls die niederwertigen 8 Bit ungleich 0 sind, liefert *_bios_keybrd* das nächste Zeichen aus dem Tastaturpuffer oder wartet auf die Eingabe eines Zeichens und liefert dieses Zeichen zurück. Bei der Eingabe eines »normalen« Zeichens liefert *_bios_keybrd* den entsprechenden ASCII-Code im niederwertigen Byte. Für Spezialzeichen wie Funktions- und Pfeiltasten hat das niederwertige Byte den Wert 0, das höherwertige Byte enthält den erweiterten Tastaturcode (siehe *Technisches Referenzhandbuch* für den IBM PC).

_NKEYBRD_READ
: – verwenden Sie diesen Wert anstelle von *_KEYBRD_READY*, wenn Zeichen einer erweiterten Tastatur abgerufen werden sollen. Eine erweiterte Tastatur verfügt über zusätzliche Pfeil- und Funktionstasten.

_KEYBRD_READY
: – prüft, ob ein Zeichen anliegt. Wenn dies nicht der Fall ist, wird der Wert 0 zurückgeliefert, ansonsten derselbe Wert wie für _KEYBRD_READ. Ein Rückgabewert von 0xFFFF(-1) bedeutet, daß [*Strg*][*Untbr*] gedrückt wurde. Über diese Funktion gelesene Zeichen werden nicht »abgeholt« – sie verbleiben im Tastaturpuf-

fer und werden durch den nächsten Aufruf von *_bios_keybrd* mit dem *cmd*-Wert _KEYBRD_READ oder _NKEYBRD_READ ausgegeben.

_NKEYBRD_READY
– verwenden Sie diesen Wert für die Überprüfung des Status einer erweiterten Tastatur. Eine erweiterte Tastatur verfügt über zusätzliche Pfeil- und Funktionstasten.

_KEYBRD_SHIFTSTATUS
– liefert den aktuellen Zustand der Umschalttasten ([*Shift*], [*Alt*] usw.). Der Rückgabewert ergibt sich aus den über ODER-Operationen verbundenen Bitwerten:

Bit 7	0x80	[*Einfg*] aktiv
Bit 6	0x40	[*Shift*] aktiv
Bit 5	0x20	[*Num*] aktiv
Bit 4	0x10	[*Rollen*] aktiv
Bit 3	0x08	[*Alt*] gedrückt
Bit 2	0x04	[*Strg*] gedrückt
Bit 1	0x02	[*Shift*] links gedrückt
Bit 0	0x01	[*Shift*] rechts gedrückt

_NKEYBRD_SHIFTSTATUS
– verwenden Sie diesen Wert anstelle von _KEYBRD_SHIFTSTATUS, um den aktuellen Status der Umschalttasten ([*Shift*], [*Alt*] usw.) von erweiterten Tastaturen festzustellen. Der Rückgabewert ergibt sich aus den über ODER-Operationen verbundenen Bitwerten von _KEYBRD_SHIFTSTATUS und den folgenden Werten:

Bit 15	0x8000	[*s-Abf*] gedrückt
Bit 14	0x4000	[*æ*] gedrückt
Bit 13	0x2000	[*NumLock*] gedrückt
Bit 12	0x1000	[*Rollen*] gedrückt
Bit 11	0x0800	[*Alt*] rechts gedrückt
Bit 10	0x0400	[*Strg*] rechts gedrückt
Bit 9	0x0200	[*Alt*] links gedrückt
Bit 8	0x0100	[*Strg*] links gedrückt

biosprint — bios.h

Funktion Kommunikation mit einem Drucker über BIOS-Interrupt.

Syntax

```
int biosprint(int cmd, int abyte, int port);
```

Beschreibung Diese Funktion verwendet den BIOS-Interrupt 0x17 zur Kommunikation mit einem angeschlossenen Drucker.

Der Wert von *port* legt fest, welche der parallelen Schnittstellen angesprochen wird: 0 steht für LPT1, 1 für LPT2, usw.

Der Wert von *cmd* legt die Art der Aktion fest:

0 Ausgabe des Zeichens in *abyte*;

1 Initialisierung der Schnittstelle;

2 Abfrage des Drucker-Status.

Der Wert von *abyte* kann zwischen 0 und 255 liegen.

Rückgabewert Alle drei Operationen liefern den aktuellen Status des angesprochenen Druckers. Der Rückgabewert ergibt sich aus den über ODER-Operationen verbundenen Bitwerten. Die Bedeutung der einzelnen Bits im gesetzten Zustand:

Bit 0	0x01	Timeout
Bit 3	0x08	I/O-Fehler
Bit 4	0x10	Drucker ist ausgewählt
Bit 5	0x20	Papierende
Bit 6	0x40	Bestätigung
Bit 7	0x80	Drucker wartet

_bios_printer bios.h

Funktion Kommunikation mit einem Drucker über BIOS-Interrupt.

Syntax

```
unsigned _bios_printer(int cmd, int port, int abyte);
```

Beschreibung Die Funktion *_bios_printer* verwendet den BIOS-Interrupt 0x17 zur Kommunikation mit einem angeschlossenen Drucker.

Der Wert von *port* legt fest, welche der parallelen Schnittstellen angesprochen wird: 0 steht für LPT1, 1 für LPT2, usw.

Der Wert von *cmd* legt die Art der Aktion fest. Folgende Werte sind in bios.h definiert:

_PRINTER_WRITE	Ausgabe des Zeichens in *abyte*; der Wert von *abyte* kann zwischen 0 und 255 liegen;
_PRINTER_INIT	Initialisierung der Schnittstelle; der Parameter *abyte* wird ignoriert;
_PRINTER_STATUS	Abfrage des Drucker-Status; der Parameter *abyte* wird ignoriert.

Rückgabewert

Alle drei Operationen liefern den aktuellen Status des angesprochenen Druckers. Der Rückgabewert ergibt sich aus den über ODER-Operationen verbundenen Bitwerten. Die Bedeutung der einzelnen Bits im gesetzten Zustand:

Bit 0	0x01	Timeout
Bit 3	0x08	I/O-Fehler
Bit 4	0x10	Drucker ist selektiert
Bit 5	0x20	Papierende
Bit 6	0x40	Bestätigung
Bit 7	0x80	Drucker wartet

_bios_serialcom bios.h

Funktion

Steuert Ein- und Ausgaben über serielle Schnittstellen.

Syntax

```
unsigned _bios_serialcom(int cmd, int port, char abyte);
```

Beschreibung

Die Funktion *_bios_serialcom* ermöglicht das Setzen von Parametern der RS-232-Schnittstellen sowie die Ein- und Ausgabe von Daten über die in *port* angegebene Schnittstelle.

Der Parameter *port* legt fest, mit welcher der Schnittstellen gearbeitet wird (0 = COM1, 1 = COM2, usw.).

Der Parameter *cmd* kann folgende in bios.h definierten Werte annehmen:

_COM_INIT	setzt die Übertragungsparameter auf den in *abyte* enthaltenen Wert;
_COM_SEND	gibt den Wert von *abyte* über die serielle Schnittstelle aus;
_COM_RECEIVE	liest ein Zeichen über die serielle Schnittstelle; der Wert von *abyte* wird ignoriert;
_COM_STATUS	liest den aktuellen Status der seriellen Schnittstelle; der Wert von *abyte* wird ignoriert.

Wenn der Parameter *cmd* den Wert _COM_INIT hat, wird *abyte* als ODER-Kombination von Bitwerten interpretiert (dabei wird aus jeder Gruppe ein Wert ausgewählt):

	Wert	Bedeutung
Datenbits	_COM_CHR7	7 Datenbits
	_COM_CHR7	8 Datenbits

Stopbits	_COM_STOP1	1 Stopbit
	_COM_STOP2	2 Stopbits
Art der Paritätsprüfung	_COM_NOPARITY	keine Paritätsprüfung
	_COM_ODDPARITY	ungerade Parität
	_COM_EVENPARITY	gerade Parität
Übertragungsgeschwindigkeit	_COM_110	110 Baud
	_COM_150	150 Baud
	_COM_300	300 Baud
	_COM_600	600 Baud
	_COM_1200	1200 Baud
	_COM_2400	2400 Baud
	_COM_4800	4800 Baud
	_COM_9600	9600 Baud

Die einzelnen Werte müssen durch eine ODER-Operation miteinander verbunden werden. Ein Beispiel: die Parameter »9600 Baud, ungerade Parität, 1 Stopbit, 8 Datenbits« werden mit dem Wert _COM_INIT von *abyte* gesetzt, der sich aus (_COM_9600 | _COM_ODDPARITY | _COM_STOP1 | _COM_CHR8) ergibt.

_bios_serialcom verwendet den BIOS-Interrupt 0x14.

Rückgabewert

Das höherwertige Byte des von *_bios_serialcom* zurückgelieferten 16-Bit-Integers enthält für alle Werte von *cmd* eine Statusinformation, die in einzelnen Bits codiert ist, das niederwertige Byte hängt von *cmd* ab. Die Bedeutung der höherwertigen Bits:

Bit 15	Time out
Bit 14	Schieberegister für Übertragung ist leer
Bit 13	Übertragungspuffer leer
Bit 12	Break-Signal erhalten
Bit 11	Framing-Fehler
Bit 10	Paritätsfehler
Bit 09	Überlauf
Bit 08	Daten bereit

Wenn das in *abyte* enthaltene Zeichen nicht gesendet werden konnte, wird Bit 15 auf 1 gesetzt. In allen anderen Fällen werden die übrigen höher- und niederwertigen Bits in geeigneter Weise gesetzt. Bei einem Framing-Fehler wird beispielsweise das Bit 11 auf 1 gesetzt.

Wenn *cmd* den Wert _COM_RECEIVE hat und ein Zeichen fehlerfrei gelesen werden konnte, enthalten die niederwertigen Bits das gelesene Zeichen. Im Fall eines Fehlers wird mindestens eines der höherwertigen Bits auf 1 gesetzt. Ist keines der höherwertigen Bits auf 1 gesetzt, wurde das Zeichen fehlerfrei gelesen.

Hat *cmd* den Wert _COM_INIT oder _COM_STATUS, liefert *_bios_serialcom* einen Wert zurück, dessen höherwertiges Byte sich aus den oben

beschriebenen Bits zusammensetzt. Zusätzlich enthält das niederwertige Byte die folgenden Informationen:

Bit 07	Received line signal detect
Bit 06	Ring indicator
Bit 05	Data set ready
Bit 04	Clear to send
Bit 03	Change in receive line signal detector
Bit 02	Trailing edge ring detector
Bit 01	Change in data set ready
Bit 00	Change in clear to send

brk — alloc.h

Funktion Ändert die Speicherbelegung für das Datensegment.

Syntax

```
int brk(void *addr);
```

Beschreibung *brk* ändert die Speichergröße, die vom Heap des aufrufenden Programms belegt wird. Die Änderung wird durchgeführt, indem der *Break*-Wert des Programms neu gesetzt wird. Der *Break*-Wert ist die Adresse der ersten Speicherstelle nach dem Ende des Datensegments. Durch Inkrementieren des *Break*-Wertes wird die Größe des belegten Speichers erhöht.

brk setzt den *Break*-Wert auf *addr* und ändert die Speicherbelegung entsprechend. Falls nicht genügend Speicher zur Verfügung steht, ändert die Funktion nichts an der Speicherbelegung.

Rückgabewert Nach erfolgreicher Änderung wird 0 zurückgeliefert. Im Fall eines Fehlers ist der Rückgabewert -1, das Datensegment bleibt unverändert und die globale Variable *errno* bekommt den Wert

ENOMEM Nicht genug Hauptspeicher

Siehe auch *coreleft*, *sbrk*

coreleft — alloc.h

Funktion Ermittelt, wieviele Bytes des Hauptspeichers noch unbelegt und damit verfügbar sind.

Syntax

```
unsigned coreleft(void);     /* Tiny, Small, Medium */
unsigned long coreleft(void);/* Compact, Large, Huge */
```

Beschreibung *coreleft* liefert ein Maß für die Größe des unbenutzten Speicherplatzes. Das zurückgelieferte Maß hängt vom verwendeten Speichermodell ab.

Rückgabewert *coreleft* liefert in den kleinen Datenmodellen die Größe des ungenutzen Speichers zwischen der Spitze des Heap und dem Stack zurück. In den großen Datenmodellen liefert *coreleft* die Größe zwischen der höchsten verfügbaren Speicheradresse und der Endadresse des höchsten belegten Blocks zurück.

Siehe auch *allocmem*, *brk*, *farcoreleft*, *malloc*

delay — dos.h

Funktion Verzögert die Ausführung um die angegebene Anzahl von Millisekunden.

Syntax

```
void delay(unsigned milliseconds);
```

Beschreibung Mit einem Aufruf von *delay* wird die Ausführung des laufenden Programms für die mit dem Argument *milliseconds* angegebene Zahl von Millisekunden unterbrochen. Die Durchführung einer Kalibrierung vor dem Aufruf von *delay* ist nicht mehr erforderlich. *delay* arbeitet auf die Millisekunde genau.

Rückgabewert Keiner.

Siehe auch *nosound*, *sleep*, *sound*

farcoreleft — alloc.h

Funktion Ermittelt den unbenutzten Speicherplatz auf dem FAR-Heap.

Syntax

```
unsigned long farcoreleft(void);
```

Beschreibung *farcoreleft* ermittelt, wieviel Speicherplatz auf dem FAR-Heap oberhalb des obersten belegten Blocks zur Verfügung steht.

Im Modell *Tiny* kann *farcoreleft* nicht verwendet werden.

Rückgabewert *farcoreleft* liefert die Größe des noch verfügbaren Speicherplatzes zurück. Ermittelt wird die Differenz zwischen der Endadresse des höchsten momentan belegten Blocks und dem Ende des verfügbaren Speichers.

Siehe auch *coreleft*, *farcalloc*, *farmalloc*

farheapcheck — alloc.h

Funktion Überprüft den FAR-Heap.

Syntax

```
int farheapcheck(void);
```

Beschreibung *farheapcheck* überprüft jeden Block des FAR-Heap auf seine Zeiger, Größe und weitere wichtige Attribute.

Rückgabewert Bei fehlerfreier Ausführung wird ein Wert größer als Null, im Fall eines Fehlers ein Wert kleiner als Null zurückgeliefert. Im einzelnen können für die Funktion folgende Werte zurückgeliefert werden:

_HEAPEMPTY	Es gibt keinen FAR-Heap (Wert 1)
_HEAPOK	Kein Fehler gefunden (Wert 2)
_HEAPCORRUPT	Der FAR-Heap ist ungültig (Wert -1).

Siehe auch *heapcheck*

farheapcheckfree alloc.h

Funktion Überprüft die freien Blöcke auf dem FAR-Heap auf einen konstanten Wert.

Syntax

```
int farheapcheckfree(unsigned int fillvalue);
```

Beschreibung *farheapcheckfree* überprüft die freien Blöcke des FAR-Heap auf einen konstanten Füllwert.

Rückgabewert Bei fehlerfreier Ausführung ist der Rückgabewert größer als Null, im Fall eines Fehlers kleiner als Null.

Im einzelnen können für die Funktion folgende Werte zurückgeliefert werden:

_HEAPEMPTY	Es gibt keinen FAR-Heap (Wert 1)
_HEAPOK	Kein Fehler gefunden (Wert 2)
_HEAPCORRUPT	Der FAR-Heap ist ungültig (Wert -1)
_BADVALUE	Ein anderer Wert als der Füllwert wurde gefunden (Wert -3)

Siehe auch *farheapfillfree*, *heapcheckfree*

farheapchecknode alloc.h

Funktion Überprüft einen einzelnen Knoten auf dem FAR-Heap.

Syntax

```
int farheapchecknode(void *node);
```

Beschreibung *farheapchecknode* überprüft die einzelnen Knoten auf dem FAR-Heap. Wird *farheapchecknode* nach der Freigabe eines Knotens mit einem Zeiger auf diesen Block aufgerufen, kann _BADNODE anstelle des erwarteten _FREEENTRY zurückgeliefert werden. Der Grund liegt darin, daß be-

nachbarte freie Blöcke auf dem FAR-Heap zusammengefaßt wurden und der betreffende Block deshalb nicht mehr existiert.

Rückgabewert Bei fehlerfreier Ausführung ist der Rückgabewert größer als Null, im Fall eines Fehlers kleiner als Null. Im einzelnen können für die Funktion folgende Werte zurückgeliefert werden:

_HEAPEMPTY	Es gibt keinen FAR-Heap (Wert 1)
_HEAPCORRUPT	Der FAR-Heap ist ungültig (Wert -1)
_BADNODE	Der Knoten wurde nicht gefunden (Wert -2)
_FREEENTRY	Der Knoten ist ein freier Block (Wert 3)
_USEDENTRY	Der Knoten ist ein belegter Block (Wert 4)

Siehe auch *heapchecknode*

farheapfillfree — alloc.h

Funktion Füllt die freien Blöcke des FAR-Heap mit einem konstanten Wert.

Syntax

```
int farheapfillfree(unsigned int fillvalue);
```

Beschreibung *farheapfillfree* füllt die freien Blöcke des FAR-Heap mit einem konstanten Wert.

Rückgabewert Bei fehlerfreier Ausführung ist der Rückgabewert größer als Null, im Fall eines Fehlers kleiner als Null. Im einzelnen können für die Funktion folgende Werte zurückgeliefert werden:

_HEAPEMPTY	Es gibt keinen FAR-Heap (Wert 1)
_HEAPOK	Kein Fehler gefunden (Wert 2)
_HEAPCORRUPT	Der FAR-Heap ist ungültig (Wert -1)

Siehe auch *farheapcheckfree*, *heapfillfree*

farheapwalk — alloc.h

Funktion Durchwandert den FAR-Heap Knoten für Knoten.

Syntax

```
int farheapwalk(struct farheapinfo *hi);
```

Beschreibung *farheapwalk* durchwandert den FAR-Heap Knoten für Knoten. Die Funktion geht davon aus, daß der FAR-Heap gültig ist (überprüfen Sie dies vorher mit *farheapcheck*). Mit dem letzten Block wird _HEAPOK zurückgeliefert. Bei einem erneuten Aufruf von *farheapwalk* liefert die Funktion _HEAPEND.

farheapwalk erhält einen Zeiger auf eine Struktur vom Typ *farheapinfo* (definiert in alloc.h). Setzen Sie das Feld *hi.ptr* beim ersten Aufruf der Funkti-

on auf Null. Nach der Rückkehr enthält *hi.ptr* die Adresse des ersten Blocks, während *hi.size* die Größe des Blocks in Bytes angibt. Ist der Block gerade belegt, wird das Flag *hi.in_use* gesetzt.

Rückgabewert Folgende Werte können zurückgeliefert werden:

_HEAPEMPTY	Es gibt keinen FAR-Heap (Wert 1)
_HEAPOK	Der *farheapinfo*-Block enthält gültige Daten (Wert 2)
_HEAPEND	Das Ende des FAR-Heap wurde erreicht (Wert 5)

Siehe auch *heapwalk*

freemem, _dos_freemem — dos.h

Funktion Gibt einen vorher belegten DOS-Speicherblock wieder frei.

Syntax

```
int freemem(unsigned segx);
unsigned _dos_freemem(unsigned segx);
```

Beschreibung *freemem* gibt einen zuvor mit *allocmem* belegten Speicherblock wieder frei. *_dos_freemem* gibt einen zuvor mit *_dos_allocmem* belegten Speicherblock wieder frei. Die Segmentadresse des Blocks wird über den Parameter *segx* angegeben.

Rückgabewert *freemem* und *_dos_freemem* liefern bei fehlerfreier Ausführung den Wert 0 zurückt.

Im Fall eines Fehlers liefert *freemem* den Wert -1 und *_dos_freemem* den DOS-Fehlercode zurück. Die globale Variable *errno* erhält folgenden Wert:

ENOMEM Nicht genug Platz im Hauptspeicher

Siehe auch *allocmem*, *_dos_allocmem*, *free*

harderr, hardresume, hardretn — dos.h

Funktion Installiert eine Routine zur Bearbeitung von Hardware-Fehlern und ruft diese auf.

Syntax

```
void harderr(int (*handler)());
void hardresume(int axret);
void hardretn(int retn);
```

Beschreibung Die von *harderr* gesetzte Fehlerbehandlungsroutine kann *hardresume* aufrufen, um zu DOS zurückzukehren. Der von *hardresume* zurückgelieferte Ergebniscode *rescode* enthält entweder den Abort- (2), Retry- (1) oder Ignore-Indikator (0). Der Abbruch wird über einen Aufruf des DOS-Interrupts 0x23 (Strg-Untbr) durchgeführt.

Die von *harderr* installierte Fehlerbehandlungsroutine kann mit einem Aufruf von *hardretn* direkt zum Anwendungsprogramm zurückkehren. Der Rückgabewert der DOS-Funktion, die den Hardware-Fehler auslöst, ist der Wert, der an *hardretn* übergeben wurde.

harderr installiert für das aktuelle Programm eine Hardware-Fehlerbehandlungsroutine. Diese Routine wird bei jedem Auftreten des Interrupt 0x24 aufgerufen. Eine Beschreibung dieses Interrupts finden Sie in Ihrem DOS *Referenzhandbuch*.

Die Funktion, auf die *handler* zeigt, wird beim Auftreten eines solchen Interrupts mit den folgenden Argumenten aufgerufen:

```
handler(int errval, int ax, int bp, int si);
```

errval ist der von DOS im DI-Register übergebene Fehlercode. *ax*, *bp* und *si* enthalten die Werte der Register AX, BP und SI:

- über den Wert von *ax* wird zwischen Gerätefehlern (*ax = 0*) und Laufwerksfehlern (*ax* <> 0) unterschieden. Wenn *ax* einen Wert ungleich Null hat, dann ergibt die Operation *ax* ver-UND-et mit *0x00FF* die Nummer des Laufwerks, das gescheitert ist. (A = 0, B = 1 etc.);
- die Parameter *bp* und *si* zeigen zusammen auf den Header des Gerätetreibers, in dem der Fehler aufgetreten ist bzw. bemerkt wurde. *bp* enthält die Segmentadresse, *si* den Offset.

Die Funktion, auf die *handler* zeigt, wird nicht direkt aufgerufen. *harderr* installiert einen DOS-Interrupt-Handler, der diese Funktion aufruft.

Die DOS-Funktionen 0x01 bis 0x0C stehen auch innerhalb der Fehlerbehandlungsroutine zur Verfügung – alle anderen Aufrufe von DOS aus bringen dagegen das System zum Absturz. Insbesondere die Standardfunktionen von C zur Ein- und Ausgabe sowie die UNIX-Emulationen dürfen nicht verwendet werden.

Der Rückgabewert von *handler* muß 0 für »Ignorieren«, 1 für »Wiederholen« und 2 für »Abbrechen« sein.

Rückgabewert Keiner.

Siehe auch *peek*, *poke*

_harderr dos.h

Funktion Installiert eine Routine zur Behandlung von Hardware-Fehlern.

Syntax

```
void _harderr(int (far *handler)());
```

Beschreibung *_harderr* installiert für das aktuelle Programm eine Hardware-Fehlerbehandlungsroutine. Diese Routine wird bei jedem Auftreten eines Interrupt 0x24 aufgerufen. Eine Beschreibung dieses Interrupt finden Sie in Ihrem DOS-Referenzhandbuch.

Die Funktion, auf die *handler* zeigt, wird beim Auftreten eines solchen Interrupt mit den folgenden Argumenten aufgerufen:

```
void far handler(unsigned deverr, unsigned errval, unsigned far *devhdr);
```

- *deverr* ist der von DOS im AX-Register übergebene Geräte-Fehlercode;
- *errval* ist der von DOS im DI-Register übergebene Fehlercode;
- *devhdr* ist ein `far`-Zeiger auf den Header des Gerätetreibers, in dem der Fehler aufgetreten ist bzw. bemerkt wurde (er wird von DOS im Registerpaar BP:SI übergeben).

Beziehen Sie sich in der Fehlerbehandlungsroutine nicht direkt auf CPU-Register, sondern verwenden Sie die oben aufgeführten Argumente.

Über den Wert von *deverr* wird zwischen Geräte- und Diskettenfehlern unterschieden. Wenn das Bit 15 von *deverr* 0 ist, hat sich ein Diskettenfehler ereignet. Andernfalls handelt es sich um einen Gerätefehler. Bei einem Diskettenfehler ergibt die Operation *deverr & 0x00FF* die Nummer des entsprechenden Laufwerks (A = 0, B = 1 etc.).

Die Funktion, auf die *handler* zeigt, wird nicht direkt aufgerufen. *_harderr* installiert einen DOS-Interrupt-Handler, der diese Funktion aufruft.

Die DOS-Funktionen 0x01 bis 0x0C stehen auch innerhalb der Fehlerbehandlungsroutine zur Verfügung – alle anderen Aufrufe von DOS bringen dagegen das System zum Absturz. Insbesondere die Standardfunktionen von C zur Ein- und Ausgabe sowie die UNIX-Emulationen dürfen nicht verwendet werden. Die Fehlerbehandlungsroutine kann entweder über *_hardresume* oder über *_hardretn* beendet werden.

Rückgabewert Keiner.

Siehe auch *_hardresume*, *_hardretn*

_hardresume dos.h

Funktion Hardware-Fehlerbehandlungsroutine.

Syntax

```
void _hardresume(int rescode);
```

Beschreibung Die von *_harderr* gesetzte Fehlerbehandlungsroutine kann *_hardresume* aufrufen, um zu DOS zurückzukehren.

Der von *_hardresume* zurückgelieferte Ergebniscode *rescode* enthält einen der folgenden Werte:

_HARDERR_ABORT	Abbruch wird über einen Aufruf des DOS-Interrupt 0x23 (Strg-Untbr) durchgeführt.
_HARDERR_IGNORE	Fehler wird ignoriert.
_HARDERR_RETRY	Operation wird wiederholt.
_HARDERR_FAIL	Operation erfolglos.

Rückgabewert *_hardresume* hat keinen Rückgabewert und kehrt nicht zur aufrufenden Funktion zurück.

Siehe auch *_harderr*, *_hardretn*

_hardretn dos.h

Funktion Hardware-Fehlerbehandlungsroutine.

Syntax

```
void _hardretn(int retn);
```

Beschreibung Die von *_harderr* installierte Fehlerbehandlungsroutine kann mit einem Aufruf von *_hardretn* direkt zum Anwendungsprogramm zurückkehren.

DOS-Funktionen unter 0x38, die einen Hardware-Fehler auslösen und die Fehlerbedingung anzeigen, setzen bei der Rückkehr zum Anwendungsprogramm das Register AL auf 0xFF.

Das Argument *retn* wird bei allen DOS-Funktionen unter 0x38 ignoriert.

Für DOS-Funktionen größer oder gleich 0x38 sollte das Argument *retn* ein DOS-Fehlercode sein. Er wird bei der Rückkehr zum Anwendungsprogramm im AX-Register übergeben. Das Carry-Flag ist gesetzt und zeigt an, daß die Operation einen Fehler zur Folge hatte.

Rückgabewert *_hardretn* hat keinen Rückgabewert und kehrt nicht zur aufrufenden Funktion zurück.

Siehe auch *_harderr*, *_hardresume*

keep, _dos_keep dos.h

Funktion Beendet ein Programm und macht es speicherresident.

Syntax

```
void keep(unsigned char status, unsigned size);
void _dos_keep(unsigned char status, unsigned size);
```

Beschreibung *keep* und *_dos_keep* verwenden die DOS-Funktion 0x31, um ein laufendes Programm zu beenden und den durch *status* angegebenen Wert als Exit-Code zurückzuliefern. Das beendete Programm bleibt resident und wird auf die Länge der über *size* angegebenen Anzahl von Paragraphen (Einheiten zu je 16 Byte) gesetzt, der Rest des vom Programm belegten Speichers wird freigegeben.

Verwenden Sie *keep* und *_dos_keep* bei der Installation von TSR-Programmen.

Bevor *_dos_keep* das laufende Programm beendet, werden alle mit *atexit* festgelegten Exit-Funktionen aufgerufen, Dateipuffer entleert und durch den Startup-Code modifizierte Interrupt-Vektoren wiederhergestellt.

Rückgabewert Keiner.

Siehe auch *abort*, *exit*

nosound — dos.h

Funktion Schaltet den eingebauten Lautsprecher ab.

Syntax

```
void nosound(void);
```

Beschreibung Ein mit *sound* eingeschalteter Lautsprecher kann durch einen Aufruf von *nosound* wieder abgeschaltet werden.

Rückgabewert Keiner.

Siehe auch *delay*, *sound*

_OvrInitEms — dos.h

Funktion Initialisiert den Expansionsspeicher für die Nutzung durch die Overlay-Speicherverwaltung.

Syntax

```
int cdecl far _OvrInitEms(unsigned emsHandle, unsigned firstPage,
                          unsigned pages);
```

Beschreibung *_OvrInitEms* prüft, ob Expansionsspeicher vorhanden ist, indem es einen EMS-Treiber sucht und von ihm Speicher reservieren läßt.

Ist *emsHandle* gleich Null, reserviert die Overlay-Speicherverwaltung EMS-Seiten und verwendet sie zur Auslagerung von Daten.

Ist *emsHandle* nicht Null, so dürfte es ein gültiges EMS-Handle sein, das die Overlay-Speicherverwaltung zur Datenauslagerung verwenden

kann. In diesem Fall können Sie mit *firstPage* die erste Seite innerhalb dieses Bereichs angeben, die zur Auslagerung verwendet werden soll.

In beiden Fällen legt ein *pages*-Argument ungleich Null die Höchstzahl der Seiten fest, die von der Overlay-Speicherverwaltung genutzt werden sollen.

Rückgabewert *_OvrInitEms* liefert 0 zurück, wenn die Overlay-Speicherverwaltung den Expansionsspeicher nutzen kann.

Siehe auch *_OvrInitExt*, *_ovrbuffer* (globale Variable)

_OvrInitExt — dos.h

Funktion Initialisiert Extended Memory für die Nutzung durch die Overlay-Speicherverwaltung.

Syntax

```
int cdecl far _OvrInitExt(unsigned long startAddress, unsigned long length);
```

Beschreibung *_OvrInitExt* prüft, ob Extended Memory vorhanden ist (d.h. die Funktion sucht mit den bekannten Methoden nach anderen Programmen, die Extended Memory benutzen), und reserviert Speicher.

Ist *startAddress* Null, bestimmt die Overlay-Speicherverwaltung die Startadresse und verwendet höchstens die Größe der Overlays. Ist *startAddress* ungleich Null, verwendet die Overlay-Speicherverwaltung Extended Memory oberhalb dieser Adresse.

In beiden Fällen legt ein *length*-Argument ungleich Null den Anteil an Extended Memory fest, der maximal von der Overlay-Speicherverwaltung genutzt werden soll.

Rückgabewert *_OvrInitExt* liefert 0 zurück, wenn die Overlay-Speicherverwaltung Extended Memory nutzen kann.

Siehe auch *_OvrInitEms*, *_ovrbuffer* (globale Variable)

randbrd — dos.h

Funktion Liest Records einer Random-Datei.

Syntax

```
int randbrd(struct fcb *fcb, int rcnt);
```

Beschreibung *randbrd* verwendet die DOS-Funktion 0x27, um die durch *rcnt* angegebene Anzahl von Records einer Datei in den Speicher zu lesen. Die Datei wird über einen »file control block« (FCB) bezeichnet, auf den *fcb* zeigt, und muß zum Zeitpunkt des Aufrufs von *randbrd* bereits offen sein; gele-

sene Blöcke werden ab der Adresse gespeichert, die durch die aktuelle *disk transfer address* (DTA) gesetzt ist.

Die tatsächliche Zahl gelesener Records kann dem Random-Record-Feld des FCB entnommen werden. Dieses Feld wird um die Zahl der tatsächlich gelesenen Records erhöht.

Rückgabewert *randbrd* liefert Werte im Bereich von 0 bis 3 zurück:

- 0 Die durch *rcnt* angegebene Anzahl von Records konnte fehlerfrei gelesen werden.
- 1 Dateiende erreicht, der letzte Record wurde vollständig gelesen.
- 2 Das Lesen von Records würde über eine Segmentgrenze im Speicher hinausgehen (es werden so viele Records wie möglich bearbeitet, danach folgt der Abbruch).
- 3 Dateiende erreicht, der letzte gelesene Record ist nicht vollständig.

Siehe auch *getdta*, *randbwr*, *setdta*

randbwr dos.h

Funktion Schreibt Records in eine Random-Datei.

Syntax

```
int randbwr(struct fcb *fcb, int rcnt);
```

Beschreibung *randbwr* verwendet die DOS-Funktion 0x28, um die durch *rcnt* angegebene Anzahl von Records in eine Datei zu schreiben. Die Datei wird über einen »file control block« (FCB) bezeichnet, auf den *fcb* zeigt, und muß zum Zeitpunkt des Aufrufs von *randbwr* bereits offen sein.

Durch die Angabe von 0 für *rcnt* wird die Datei auf die im Random-Record-Feld des FCB angegebene Länge verkürzt.

Die tatsächliche Zahl geschriebener Records kann dem Random-Record-Feld des FCB entnommen werden. Dieses Feld wird um die Zahl der tatsächlich geschriebenen Records erhöht.

Rückgabewert *randbwr* liefert Werte im Bereich von 0 bis 2 zurück:

- 0 Die durch *rcnt* angegebene Anzahl von Records konnte fehlerfrei geschrieben werden.
- 1 Nicht genügend Platz auf der Diskette (diese Prüfung erfolgt *vor* dem Schreiben des ersten Records – wenn nicht genügend Platz vorhanden ist, bricht *randbwr* sofort ab).

2 Das Schreiben von Records würde über eine Segmentgrenze im Speicher hinausgehen (es werden so viele Records wie möglich bearbeitet, danach folgt der Abbruch).

Siehe auch *randbrd*

sbrk — alloc.h

Funktion Verändert die Größe des Datensegments.

Syntax

```
void *sbrk(int incr);
```

Beschreibung *sbrk* fügt *incr* Bytes zum *Break*-Wert des Programms hinzu und ändert die belegte Speichergröße entsprechend. Wenn *incr* ein negativer Wert ist, wird der belegte Speicher verkleinert.

sbrk liefert einen Fehler zurück und ändert nichts an der Speicherbelegung, wenn durch die Änderung mehr Speicher belegt würde, als erlaubt ist.

Rückgabewert Im Fehlerfall wird der Wert -1 zurückgeliefert, das Datensegment bleibt unverändert und die globale Variable *errno* bekommt den Wert

ENOMEM Nicht genug Platz im Hauptspeicher

Die fehlerfreie Ausführung von *sbrk* liefert den vorherigen *Break*-Wert zurück.

Siehe auch *brk*

setblock, _dos_setblock — dos.h

Funktion Verändert die Größe eines zuvor belegten Speicherblocks.

Syntax

```
int setblock(unsigned segx, unsigned newsize);
unsigned _dos_setblock(unsigned newsize, unsigned segx, unsigned *maxp);
```

Beschreibung *setblock* und *_dos_setblock* verändern die Größe eines zuvor mit *allocmem* bzw. *_dos_allocmem* belegten Speichersegments. Die Startadresse des Blocks wird über den Parameter *segx* angegeben, *newsize* gibt die neue Größe in Paragraphen an.

Kann der Speicherblock nicht auf die angegebene Größe geändert werden, speichert *_dos_setblock* den Umfang des größtmöglichen Bereichs an der durch *maxp* angegebenen Adresse.

Rückgabewert Wenn ein Speicherbereich der gewünschten Größe belegt werden konnte, liefert *setblock* den Wert -1. Wenn nicht, wird die Größe des größten noch

verfügbaren Speicherblocks (in Paragraphen) zurückgeliefert und die globale Variable *_doserrno* gesetzt.

_dos_setblock hat bei fehlerfreier Ausführung den Rückgabewert 0. Im Falle eines Fehlers wird der DOS-Fehlercode zurückgeliefert und die globale Variable *errno* wird auf folgenden Wert gesetzt:

ENOMEM	Nicht genug Platz im Hauptspeicher oder falsche Segementadresse

Siehe auch *allocmem, freemem*

sound — dos.h

Funktion Aktiviert den eingebauten Lautsprecher.

Syntax

```
void sound(unsigned frequency);
```

Beschreibung *sound* erzeugt einen Ton im eingebauten Lautsprecher, dessen Frequenz über den Wert von *frequency* in Hertz (Schwingungen pro Sekunde) festgelegt ist. Ein mit *sound* gestarteter Ton wird so lange erzeugt, bis der Lautsprecher durch einen Aufruf von *nosound* wieder abgeschaltet wird.

Rückgabewert Keiner.

Siehe auche *delay, nosound*

A N H A N G A

DOS-Bibliotheken

In diesem Anhang werden die für 16-Bit DOS-Anwendungen verfügbaren Bibliotheksroutinen von Borland C++ im Überblick aufgelistet. Die Bibliotheksroutinen setzen sich aus Klassen, Funktionen und Makros zusammen, die Sie innerhalb Ihrer C und C++ Programme aufrufen können, um die eine große Vielfalt von Aufgaben zu übernehmen. Dazu gehören beispielsweise Ein-/Ausgabe, String- und Dateibehandlung, Speicherplatzreservierung, Prozeßsteuerung, Datenkonvertierung, mathematische Berechnungen und vieles mehr.

Dieses Kapitel enthält die folgenden Informationen:

- Die Namen der Bibliotheken und Dateien, die Sie im Unterverzeichnis LIB finden, werden aufgelistet und ihre Verwendung wird beschrieben.
- Die Bibliotheksroutinen werden – nach Aufgabenbereichen in Kategorien eingeteilt – aufgelistet.

Die Laufzeitbibliotheken

DOS-Anwendungen benutzen statische Laufzeitbibliotheken (OBJ und LIB). Die hier aufgeführten Bibliotheken stehen nur für 16-Bit Entwicklungen zur Verfügung. Im Kapitel 1 des *Referenzhandbuchs* finden Sie eine Beschreibung der übrigen Bibliotheken.

Es gibt verschiedene Versionen der Laufzeitbibliotheken, beispielsweise verschiedene Versionen für jedes Speichermodell und spezielle Versionen für Diagnosezwecke.

Weiterhin existieren Bibliotheken, die Containerklassen, Grafik- und Mathematikfunktionen unterstützen.

Hier einige Hinweise zur Auswahl der Laufzeitbibliotheken:

- Die weiter unten aufgeführten Bibliotheken stehen nur für 16-Bit DOS-Anwendungen zur Verfügung.
- Informationen über zusätzliche DOS-Routinen finden Sie im *Referenzhandbuch*.
- Exception-Behandlung darf bei Verwendung von Overlays nicht benutzt werden.

Die DOS-Bibliotheken

Die statische Version (OBJ und LIB) der Borland C++ Laufzeitbibliothek befindet sich im Unterverzeichnis LIB Ihrer Installation. Bei den folgenden Dateinamen vertritt das Fragezeichen ('?') jeweils eines der sechs von Borland unterstützten Speichermodelle. Mit Ausnahme des Speichermodells Tiny besitzt jedes Modell seine eigene Bibliotheks- und Mathematikdatei, die die Version der Routine enthält, die für das entsprechende Speichermodell entwickelt wurden. Im Kapitel 1 finden Sie Einzelheiten zu Speichermodellen.

Die folgende Tabelle enthält die Namen der Borland C++ Bibliotheken, die ausschließlich für 16-Bit DOS-Anwendungen verfügbar sind. Im *Benutzerhandbuch* finden Sie weitere Informationen über Linken, Linkeroptionen, Anforderungen und Auswahl der Bibliotheken. Im Kapitel 1 des *Referenzhandbuchs* finden Sie Informationen über die übrigen Bibliotheken und Header-Dateien, die teilweise ebenfalls in DOS-Anwendungen benutzt werden können.

Tabelle A.1: Zusammenfassung der DOS-Laufzeitbibliotheken

Dateiname	Verwendung
BIDSH.LIB	BIDS-Klassenbibliothek für Speichermodell Huge
BIDSDBH.LIB	Diagnose-Version der obigen Bibliothek
C?.LIB	Bibliothek, die nur unter DOS benutzt werden darf
C0F?.OBJ	MS-kompatibler Startcode
C0?.OBJ	Borland-kompatibler Startcode
EMU.LIB	Gleitkomma-Emulation
FP87.LIB	Für Programme die mit Koprozessor laufen
GRAPHICS.LIB	BGI (Borland Graphics Interface)
MATH?.LIB	Mathematische Routinen
OVERLAY.LIB	Overlay-Entwicklung

Grafikroutinen

Mit diesen Routinen können Sie Bildschirmgrafiken mit Textdarstellungen erzeugen.

arc	(graphics.h)	*graphresult*	(graphics.h)
bar	(graphics.h)	*imagesize*	(graphics.h)
bar3d	(graphics.h)	*initgraph*	(graphics.h)
circle	(graphics.h)	*installuserdriver*	(graphics.h)

cleardevice	(graphics.h)	*installuserfont*	(graphics.h)
clearviewport	(graphics.h)	*line*	(graphics.h)
closegraph	(graphics.h)	*linerel*	(graphics.h)
detectgraph	(graphics.h)	*lineto*	(graphics.h)
drawpoly	(graphics.h)	*moverel*	(graphics.h)
ellipse	(graphics.h)	*moveto*	(graphics.h)
fillellipse	(graphics.h)	*outtext*	(graphics.h)
fillpoly	(graphics.h)	*outtextxy*	(graphics.h)
floodfill	(graphics.h)	*pieslice*	(graphics.h)
getarccoords	(graphics.h)	*putimage*	(graphics.h)
getaspectratio	(graphics.h)	*putpixel*	(graphics.h)
getbkcolor	(graphics.h)	*rectangle*	(graphics.h)
getcolor	(graphics.h)	*registerbgidriver*	(graphics.h)
getdefaultpalette	(graphics.h)	*registerbgifont*	(graphics.h)
getdrivername	(graphics.h)	*restorecrtmode*	(graphics.h)
getfillpattern	(graphics.h)	*sector*	(graphics.h)
getfillsettings	(graphics.h)	*setactivepage*	(graphics.h)
getgraphmode	(graphics.h)	*setallpalette*	(graphics.h)
getimage	(graphics.h)	*setaspectratio*	(graphics.h)
getlinesettings	(graphics.h)	*setbkcolor*	(graphics.h)
getmaxcolor	(graphics.h)	*setcolor*	(graphics.h)
getmaxmode	(graphics.h)	*_setcursortype*	(conio.h)
getmaxx	(graphics.h)	*setfillpattern*	(graphics.h)
getmaxy	(graphics.h)	*setfillstyle*	(graphics.h)
getmodename	(graphics.h)	*setgraphbufsize*	(graphics.h)
getmoderange	(graphics.h)	*setgraphmode*	(graphics.h)
getpalette	(graphics.h)	*setlinestyle*	(graphics.h)
getpalettesize	(graphics.h)	*setpalette*	(graphics.h)
getpixel	(graphics.h)	*setrgbpalette*	(graphics.h)
gettextsettings	(graphics.h)	*settextjustify*	(graphics.h)
getviewsettings	(graphics.h)	*settextstyle*	(graphics.h)
getx	(graphics.h)	*setusercharsize*	(graphics.h)
gety	(graphics.h)	*setviewport*	(graphics.h)
graphdefaults	(graphics.h)	*setvisualpage*	(graphics.h)
grapherrormsg	(graphics.h)	*setwritemode*	(graphics.h)
_graphfreemem	(graphics.h)	*textheight*	(graphics.h)
_graphgetmem	(graphics.h)	*textwidth*	(graphics.h)

Schnittstellenroutinen

Diese Routinen ermöglichen den Zugriff auf das Betriebssystem, das BIOS und die rechnerspezifischen Fähigkeiten.

absread	(dos.h)	*_dos_freemem*	(dos.h)
abswrite	(dos.h)	*freemem*	(dos.h)
bioscom	(bios.h)	*_harderr*	(dos.h)
_bios_disk	(bios.h)	*harderr*	(dos.h)
biosdisk	(bios.h)	*_hardresume*	(dos.h)
_bios_keybrd	(bios.h)	*hardresume*	(dos.h)
bioskey	(bios.h)	*_hardretn*	(dos.h)
biosprint	(bios.h)	*hardretn*	(dos.h)

_bios_printer	(bios.h)	*keep*	(dos.h)
_bios_serialcom	(bios.h)	*randbrd*	(dos.h)
_dos_keep	(dos.h)	*randbwr*	(dos.h)

Routinen zur Speicherverwaltung

Die Routinen ermöglichen die dynamische Speicherreservierung in den Speichermodellen *Small* und *Large*.

allocmem	(dos.h)	*farheapcheck*	(alloc.h)
brk	(alloc.h)	*farheapcheckfree*	(alloc.h)
coreleft	(alloc.h, stdlib.h)	*farheapchecknode*	(alloc.h)
_dos_allocmem	(dos.h)	*farheapfillfree*	(alloc.h)
_dos_freemem	(dos.h)	*farheapwalk*	(alloc.h)
_dos_setblock	(dos.h)	*farrealloc*	(alloc.h)
farcoreleft	(alloc.h)	*sbrk*	(alloc.h)

Verschiedene Routinen

Diese Routinen implementieren Geräuscheffekte und Zeitverzögerungen.

delay	(dos.h)	*sound*	(dos.h)
nosound	(dos.h)		

A N H A N G B

Globale Variablen für DOS-Anwendungen

In diesem Anhang werden die globalen Variablen von Borland C++ beschrieben, die nur für 16-Bit DOS-Anwendungen zur Verfügung stehen. Weitere globale Variablen finden Sie im Kapitel 4 des *Referenzhandbuchs*.

_heaplen — dos.h

Funktion Enthält die Größe des Near-Heap.

Syntax

```
extern unsigned _heaplen;
```

Deklaration in dos.h

Beschreibung *_heaplen* enthält die Größe des Near-Heap in Bytes und existiert nur in den kleinen Speichermodellen (*Tiny*, *Small* und *Medium*); in den größeren Speichermodellen (*Compact*, *Large* und *Huge*) gibt es keinen Near-Heap.

In den Modellen *Small* und *Medium* wird die Größe des Datensegments wie folgt berechnet:

```
Datensegment [small, medium] = globale Daten + Heap + Stack
```

Die Größe des Stack kann mit *_stklen* angepaßt werden.

Wenn *_heaplen* auf 0 gesetzt wird, reserviert das Programm 64 KByte des Speichers für das Datensegment, die effektive Größe des Heap beträgt dann:

```
64 KByte - (Globale Daten + Stack) Bytes
```

Die Standardgröße für *_heaplen* ist 0. Wird daher kein Wert für *_heaplen* angegeben, erhält man ein 64 KByte großes Datensegment.

Im Modell *Tiny* befindet sich alles (auch der Programmcode) in demselben Segment, so daß die Größe des Programmcodes plus 256 Bytes für das Programmsegmentpräfix (PSP) für die Berechnung der Datensegmentgröße berücksichtigt wird:

```
Datensegment [tiny] = 256 Bytes PSP + Programmcode
                    + globale Daten + Heap
                    + Stack
```

Wenn *_heaplen* im Modell *Tiny* auf 0 gesetzt wird, berechnet sich die sich die effektive Größe des Heap wie folgt:

```
Heapgröße = 64 KByte - (Programmcode + 256 Bytes PSP
                        globale Daten + Stack)
```

In den Speichermodellen *Compact* und *Large* gibt es keinen Near-Heap. Der Stack befindet sich in einem eigenen Segment, die Größe des Datensegments ist:

```
Datensegment [compact, large] = globale Daten
```

Im Speichermodell *Huge* ist der Stack ein separates Segment, jedes Modul hat hier sein eigenenes Datensegment.

Siehe auch *_stklen*

_ovrbuffer — dos.h

Funktion Ändert die Größe des Overlay-Puffers.

Syntax

```
unsigned _ovrbuffer = size;
```

Deklaration in dos.h

Beschreibung Die Standardgröße des Overlay-Puffers entspricht dem doppelten Umfang des größten Overlay. Das ist für einige Anwendungen sicher angemessen.

Wenn jedoch eine bestimmte Funktion eines Programms mit Hilfe vieler Overlay-Module implementiert wurde, dann kann es Probleme geben: Sobald die Gesamtgröße aller Module größer als der Overlay-Puffer ist, werden Module ein- und ausgelagert. Dies ist umso häufiger der Fall, wenn sich die Module bei der Ausführung der Funktion oft wechselseitig aufrufen müssen.

Zur Lösung dieses Problems kann die Größe des Overlay-Puffers erhöht werden, so daß jederzeit genügend Speicherplatz für Overlay-Module zur Verfügung steht, die sich häufig wechselseitig aufrufen. Dazu wird die globale Variable *_ovrbuffer* auf die benötigte Größe gesetzt.

Dazu ein Beispiel: Die folgende Anweisung setzt den Overlay-Puffer auf 128 KByte:

```
unsigned _ovrbuffer = 0x2000;
```

Es gibt keine Formel für die Berechnung der idealen Größe des Overlay-Puffers.

Siehe auch *_OvrinitEMS*, *_OvrinitExt*

_stklen dos.h

Funktion Enthält die Größe des Stack.

Syntax

```
extern unsigned _stklen;
```

Deklaration in dos.h

Beschreibung *_stklen* enthält die Größe des Stack in allen sechs Speichermodellen. Die Minimalgröße von *_stklen* beträgt 256 Bytes (128 WORD); kleinere Werte werden automatisch durch dieses Minimum ersetzt. Die Voreinstellung für die Größe des Stack ist 4 KByte.

In den Modellen *Small* und *Medium* wird die Größe des Datensegments wie folgt berechnet:

```
Datensegment [small, medium] = globale Daten + Heap
                             +   Stack
```

Die Größe des Heap kann mit *_heaplen* angepaßt werden.

Im Modell *Tiny* befindet sich alles (auch der Programmcode) in demselben Segment, so daß die Größe des Programmcodes plus 256 Bytes für das Programmsegmentpräfix (PSP) für die Berechnung der Datensegmentgröße berücksichtigt wird:

```
Datensegment [tiny] = 256 Bytes PSP + Programmcode
                    + globale Daten + near Heap
                    + Stack
```

In den Speichermodellen *Compact* und *Large* gibt es keinen Near-Heap, der Stack befindet sich hier in einem eigenen Segment.

Die Größe des Datensegments ist dann einfach:

```
Datensegment [compact, large] = globale Daten
```

Im Modell *Huge* ist der Stack ein separates Segment, jedes Modul hat hier sein eigenes Datensegment.

Siehe auch *_heaplen*

Index

!

!=-Operator
    Vergleich von Huge-Zeiger und *17*
#pragma-Anweisung
    Far-Objekte und die *25*
-Y Kommandozeilenoption (Durch den Compiler erzeugter Code für Overlays) *35*
-Yo - Option (Overlays) *34*
-zX-Optionen (Code- und Daten-Segment *25*
.CHR-Dateien *107*
0x11, BIOS-Interrupt *122*
0x13, BIOS-Interrupt *117*
0x16, BIOS-Interrupt *116f*
0x17, BIOS-Interrupt *118f*
0x23, DOS-Interrupt *132*, *134*
0x24, DOS-Interrupt *126f*
0x25, DOS-Interrupt *113*
0x26, DOS-Interrupt *114*
0x27, DOS-Funktion *137*
0x28, DOS-Funktion *138*
0x31, DOS-Funktion *135*
0x38, DOS-Funktion *135*
0x48, DOS-Systemaufruf *114*
80x86-Prozessor-Register *12*
80x86-Prozessoren
    Adressierungsnotation der Form Segment:Offset *15*
==-Operator
    Vergleich von Huge-Zeiger und *17*
__cs (Schlagwort) *23*
__ds (Schlagwort) *23*
__es (Schlagwort) *23*
__far (Schlagwort) *23*
__huge (Schlagwort) *23*
__near (Schlagwort) *23*
__seg (Schlagwort) *23*
__ss (Schlagwort) *23*
_bios_disk (Bibliotheksfunktion) *120*
_bios_keybrd (Bibliotheksfunktion) *123*
_bios_printer (Bibliotheksfunktion) *125*
_bios_serialcom (Bibliotheksfunktion) *126*
_dos_allocmem (Bibliotheksfunktion) *114*
_dos_freemem (Bibliotheksfunktion) *132*
_dos_keep (Bibliotheksfunktion) *135*
_dos_setblock (Bibliotheksfunktion) *139*
_graphfreemem (Bibliotheksfunktion) *84*
_graphgetmem (Bibliotheksfunktion) *84*
_harderr (Bibliotheksfunktion) *133*
_hardresume (Bibliotheksfunktion) *134*
_hardretn (Bibliotheksfunktion) *135*
_heaplen (globale Variable) *9*
_ovrbuffer (globale Variable) *10*, *31*, *35*
_OvrInitEms (Bibliotheksfunktion) *136*
_OvrInitExt (Bibliotheksfunktion) *137*
_seg (Schlagwort) *24*
_stklen (globale Variable) *11*

A

absread (Bibliotheksfunktion) *113*
abswrite (Bibliotheksfunktion) *114*
Adapter
    Grafik- *68*
aktive Grafikseite *97*
Aktive Seite
    Bestimmen der *54*
    definierte *54*
aktuelle Position (CP) *91*
    bewegen *92*
    Koordinaten *109*
    Position ermitteln *82*
    setzen *92*
allocmem (Bibliotheksfunktion) *114*
Angezeigte Seite
    definierte *54*
    Setzen der *54*
arc (Bibliotheksfunktion) *65*
    Koordinaten ermitteln *72*
arccoordstype (Datentyp) *72*
Assembler
    Overlays und Routinen in *36*
Attribut
    Bildschirmzellen *47*
Auffang-Flag *14*
Ausfüllen
    Funktionen zum *52*

Ausrichtung
von Textausgaben *107*
Auxiliary Carry-Flag *14*
AX-Register *12*

B

Balkengrafiken *66*
bar (Bibliotheksfunktion) *66*
bar3d (Bibliotheksfunktion) *66*
Basisadreßregister *13*
Baudrate *116*, *126*
BCD-Arithmetik *38ff*
Konvertierung *45*
Rundung von Zahlen *46*
bcd.h (Deklarationsdatei) *44*
BGI
Grafiktreiber und *84*, *86*, *89*
BGIOBJ (Grafikkonverter)
initgraph Funktion und *51*
Bibliotheken
Auswahl von *141*
Grafik *49*
Bildausschnitte
Ausgabe auf Bildschirm *94*
in der Grafik *76*, *86*
Bildpunkte
ermitteln *81*
setzen *95*
Bildschirm
Auflösung *48*
Ausschnitt speichern *76*, *86*
Farben *57*
Grafikkarten *50*
Höhen-/Seitenverhältnis *53*, *72*
Höhen-/Seitenverhältnis ändern *99*
Koordinaten *48*
Löschen *54*, *67*, *103*
maximale Koordinaten *72f*
maximale X-Koordinate *72f*
Modi *41f*, *44f*
Zelle *47*
Bildschirm, Grafikkarten
Grafik, kompatibel mit C++ *50*
Bildschirmadapter
Modi *47*
Bildschirmmodus
wiederherstellen *96*
BIOS-Interrupts
0x11 *122*
0x13 *117*
0x16 *116f*
0x17 *118f*
bioscom (Bibliotheksfunktion) *115*
biosdisk (Bibliotheksfunktion) *117*
bioskey (Bibliotheksfunktion) *122*
biosprint (Bibliotheksfunktion) *124*
bit images
Funktionen für *54*
BP-Register *13*
Overlays und *36*
Break-Wert *128*, *139*
Breite von Textausgaben *111*
brk (Bibliotheksfunktion) *128*
BX-Register *12*

C

Carry-Flag *14*
CGA *65*
circle (Bibliotheksfunktion) *67*
cleardevice (Bibliotheksfunktion) *67*
clearviewport (Bibliotheksfunktion) *67*
Clipping, definiertes *56*
closegraph (Bibliotheksfunktion) *68*
Code-Segment *14*
Color/Graphics Adapter (CGA)
Auflösungen *52f*
Farbpaletten *52f*
COM-Dateien
Speichermodelle und *18*
coreleft (Bibliotheksfunktion) *128*
CS-Register *14*, *16*
CX-Register *12*

D

Datei
Projekt- *49*
Dateien
.CHR *90*
FCB (file control block) *131f*
Grafiktreiber, Linken von *51*
Datenbits *116*, *126*
Datensegment *14*
Benennung und Umbenennung von *25*
Größe ändern *139*

Speicherbelegung *128*
Datentypen
struct arccoordstype *72*
struct fillsettingstype *75*
struct linesettingstype *76, 104*
struct palettetype *80, 98*
struct textsettingstype *81*
struct viewporttype *82*
Debuggen
Overlays *36*
delay (Bibliotheksfunktion) *129*
detectgraph (Bibliotheksfunktion) *68*
Dezimalzahlen
Umwandlung in Binärzahlen *38ff*
DI-Register *13*
Direction-Flag *14*
Disketten
Funktionen (_bios_disk) *120*
Funktionen (absread) *113*
Funktionen (abswrite) *114*
Funktionen (biosdisk) *117*
Diskettensektoren
lesen *113, 117, 120*
schreiben *114, 117, 120*
DMA (Direct memory access) *119*
DOS-Funktionen
0x31 *135*
0x38 *135*
DOS-Interrupts
0x23 *132, 134*
0x24 *126f*
0x25 *113*
0x26 *114*
DOS-Systemaufrufe
0x27 *137*
0x28 *138*
0x48 *114*
DOS/BIOS-Routinen
_bios_disk *120*
_bios_keybrd *123*
_bios_printer *125*
_bios_serialcom *126*
_dos_freemem *132*
_dos_keep *135*
_harderr *133*
_hardresume *134*
_hardretn *135*
absread *113*
abswrite *114*
bioscom *115*
biosdisk *117*
bioskey *122*
biosprint *124*
freemem *132*
harderr *132*
hardresume *132*
hardretn *132*
keep *135*
randbrd *137*
drawpoly (Bibliotheksfunktion) *70*
dreidimensionale Balken *66*
Drucker
direkt drucken *118f*
Schnittstelle *118f*
DS-Register *14, 16*
DTA (Disk Transfer Address) *131f*
DX-Register *12*
Dynamische Speicherverwaltung
_dos_allocmem *114*
_dos_setblock *139*
allocmem *114*
brk *128*
coreleft *128*
FAR-Heap *129*
freie Blöcke des FAR-Heap füllen *131*
Grafikspeicher belegen *84*
Grafikspeicher freigeben *84*
sbrk *139*
setblock *139*
siehe auch Speicher *114*
Überprüfung des FAR-Heap *130, 131*

E

Ein-/Ausgabe
Diskette *120*
Operationen *117*
serielle Schnittstelle *115, 126*
elleptischer Kreisbogen *70*
ellipse (Bibliotheksfunktion) *70*
Ellipsen
zeichnen und füllen *71*
EMS
siehe Expanded und Extended Memory, Speicher *97*
EMU.LIB *40*
Enhanced Graphics Adapter (EGA)
Farbsteuerung auf *60*

Environment ('Umgebung')
Variable '93' *41*
ES-Register *14*
Exception-Behandlung *33*
Exit-Code
bei keep und _dos_keep *136*
Expanded Memory (EMS) *136*
Extended und Expanded Memory
Overlays und *37*
Extendend Memory *137*
Extra-Segment *14*

F

far (Schlagwort) *16*, *28*
Far-Aufrufe
Anforderungen bei *35*
Speichermodelle und *35*
FAR-Heap
freie Blöcke füllen *131*
Knoten überprüfen *130*
überprüfen *123ff*
Überprüfen und Initialisieren *129*
ungenutzer Speicher auf *129*
Farb-Paletten
ermitteln *80*
Farben
höchste Farbnummer *77*
Farben und Paletten
Hintergrundfarbe *73*
ändern *105*
Einträge setzen *98*
Füllfarbe *71*, *94*, *97*, *102*
Füllmuster *71*, *74*, *94*, *97*, *102*
Füllmuster definieren *101*
Größe *80*
größter Farbwert für setcolor *77*
Hintergrund *100*
Informationen ermitteln *80*
Pixel *81*, *95*
Probleme *65*
Rechteck *95*
setzen *106*
Zeichen *94*
Zeichenfarbe *73*
Zeichenfarbe setzen *101*
Zeiger auf *73*
Farbgrafikadapter (CGA) *94*
Probleme mit *65*
Überprüfung auf Vorhandensein *68*
farcoreleft (Bibliotheksfunktion) *129*
farheapcheck (Bibliotheksfunktion) *129*
farheapcheckfree (Bibliotheksfunktion) *130*
farheapchecknode (Bibliotheksfunktion) *130*
farheapfillfree (Bibliotheksfunktion) *131*
farheapwalk (Bibliotheksfunktion) *131*
FCB (file control block) *131f*
Fehler
Behandlung *126ff*
Botschaft *61*
Grafik, Funktionen zum Behandeln von *61*
Hardware *127ff*
Hardware- *132*
Out of memory *11*
Fehlermeldungen
des Grafikpakets *77f*
Fehlersuche
Overlay *36*
Fenster
definiertes *48*
voreingestelltes *48*
fillellipse (Bibliotheksfunktion) *71*
fillpoly (Bibliotheksfunktion) *71*
fillsettingstype (Datentyp) *75*
Flächenfüllung *71*
Muster ermitteln *74*
Muster setzen *101*
Parameter *74*, *102*
Flag-Register *12*, *13*
floodfill (Bibliotheksfunktion) *71*
Fonts
festlegen *107*
Grafik- *81*
registrieren *96*
siehe auch Vektor-Fonts, Zeichensätze *109*
siehe auch Zeichensätze *96*
Vektor- *90*
Zeichengröße *111*
Formatierung
grafischer Textausgaben *107*
FP87.LIB *40*
freemem (Bibliotheksfunktion) *132*
Füllen
von Flächen *71*
von Polygonen *71*
Funktionen
Als near oder far deklarierende *25*
Aufruf von Overlay-Routinen über Zeiger *35*

Ausfüll- *52*
Bildmanipulation *54*
Bildschirmmanipulation *54*
Deklaration von Far *26*
Deklaration von Near- *26*
Farbsteuerung *56*
Fehlerbehandlung, Grafik *61*
Grafik *49*
Grafik- *52*
Grafik-Kontroll- des Systems *50*
Grafikfenstermanipulation *54*
Größe von Speichermodellen und Far- *25*
Speichermodelle und Near- *25*
Speichermodelle und rekursive *25*
Statusabfragen *61*
Text *55*
Zeichen- *52*

G

Genauigkeit
von Dezimalzahlen *44*
gerade Parität *115*
getarccoords (Bibliotheksfunktion) *72*
getaspectratio (Bibliotheksfunktion *72*
getbkcolor (Bibliotheksfunktion) *73*
getcolor (Bibliotheksfunktion) *73*
getdefaultpalette (Bibliotheksfunktion) *73*
getdrivername (Bibliotheksfunktion) *74*
getfillpattern (Bibliotheksfunktion) *74*
getfillsettings (Bibliotheksfunktion) *74*
getgraphmode (Bibliotheksfunktion) *76*
getimage (Bibliotheksfunktion) *76*
getlinesettings (Bibliotheksfunktion) *76*
getmaxcolor (Bibliotheksfunktion) *77*
getmaxmode (Bibliotheksfunktion) *78*
getmaxx (Bibliotheksfunktion) *78*
getmaxy (Bibliotheksfunktion) *79*
getmodename (Bibliotheksfunktion) *79*
getmoderange (Bibliotheksfunktion) *79*
getpalette (Bibliotheksfunktion) *80*
getpalettesize (Bibliotheksfunktion) *80*
getpixel (Bibliotheksfunktion) *81*
gettextsettings (Bibliotheksfunktion) *81*
getviewsettings (Bibliotheksfunktion) *82*
getx (Bibliotheksfunktion) *82*
gety (Bibliotheksfunktion) *82*
Gleitkomma-Routinen
beschleunigte *41*
Emulation *40*
Fehlerbedingungen *42*
Globale Variable
_ovrbuffer *35*
Globale Variablen
_ovrbuffer *31*
Grafik
Anzeigen von *58*
Ausfüllen *46f, 62*
Bibliothek *49*
Farben *50ff, 63*
Fehler *61*
Funktionen *49*
Grafikfenster *48, 54, 62*
Header Datei *49*
Kreise *53*
Lienienarten *53*
Palette *56*
Paletten *57, 63*
Pixel *54, 57, 62*
Puffer *54*
Seite *54*
Setzen *54*
Speicher für *52*
Standardeinstellung *51*
Statusabfragen *61*
System *44f, 62*
Text und *55*
Viewport *48*
Zeichenfunktionen *52*
Grafikadapter *68*
Monochrom *65*
Grafikkarten
Benutzung von *47*
untersuchen *86*
Grafikmodus *68, 76*
erlaubter Bereich *79*
ermitteln *76*
maximale *78*
Name *79*
setzen *103*
Grafikpaket
Beenden *68*
Fehlerbehandlung *77f*
Heap-Verwaltung *84*
Initialisierung *86*
Puffergröße *103*
Registrierung von Treibern *95*
Registrierung von Zeichensätzen *96*

Standardvorgaben *83*
Grafikpuffer
Größe *103*
Grafikroutinen
_graphfreemem *84*
_graphgetmem *84*
arc *65*
bar *66*
bar3d *66*
circle *67*
cleardevice *67*
clearviewport *67*
closegraph *68*
detectgraph *68*
drawpoly *70*
ellipse *70*
fillellipse *71*
fillpoly *71*
floodfill *71*
getarccoords *72*
getaspectratio *72*
getbkcolor *73*
getcolor *73*
getdefaultpalette *73*
getdrivername *74*
getfillpattern *74*
getfillsettings *74*
getgraphmode *76*
getimage *76*
getlinesettings *76*
getmaxcolor *77*
getmaxmode *78*
getmaxx *78*
getmaxy *79*
getmodename *79*
getmoderange *79*
getpalette *80*
getpalettesize *80*
getpixel *81*
gettextsettings *81*
getviewsettings *82*
getx *82*
gety *82*
graphdefaults *83*
grapherrormsg *83*
graphresult *84*
imagesize *86*
initgraph *86*
installuserfont *90*
line *91*
linerel *91*
lineto *91*
moverel *92*
moveto *92*
outtext *92*
outtextxy *93*
pieslice *93*
putimage *94*
putpixel *95*
Rechteck zeichnen *95*
registerbgidriver *95*
registerbgifont *96*
restorecrtmode *96*
sector *97*
setactivepage *97*
setallpalette *98*
setaspectratio *99*
setbkcolor *100*
setcolor *101*
setfillpattern *101*
setfillstyle *102*
setgraphbufsize *103*
setgraphmode *103*
setlinestyle *104*
setpalette *105*
setrgbpalette *106*
settextjustify *107*
settextstyle *107*
setusercharsize *109*
setviewport *109*
setvisualpage *110*
setwritemode *110*
textheight *111*
textwidth *111*
Grafikspeicherseite *110*
Grafiktreiber *68*
Aktuell *63*
Aktueller *51, 62*
BGI und *84, 86, 89*
Durch C++ unterstützte *50*
Erhalten von Informationen über *62*
Konstanten in graphics.h *68, 87*
laden *86*
Laden und Auswählen *44f*
Liefern von Informationen über *63*
Linken von *51*
Modus *72f, 86*
Modus setzen *96*

Modus-Ermittlung *78*
Name ermitteln *74*
Namensermittlung *74*
neuer *51, 89*
Registrieren der *51, 95*
graphdefaults (Bibliotheksfunktion) *83*
grapherrormsg (Bibliotheksfunktion) *83*
graphics.h (Header-Datei) *49*
graphresult (Bibliotheksfunktion) *84*
Größe
Datensegment *139*
Farbpalette *80*
Near-Heap *9*
Speicher *122*
von Stack und Heap *11*
von Textausgaben *107*

H

harderr (Bibliotheksfunktion) *132*
hardresume (Bibliotheksfunktion) *132*
hardretn (Bibliotheksfunktion) *132*
Hardware
Druckerschnittstelle *124, 125*
Fehlerbehandlung *126ff*
Header-Dateien
Grafik *49*
Heap *128*
FAR *123ff*
Größe festlegen *9*
Größe setzen *11*
Verwaltung durch das Grafikpaket *84*
Hintergrundfarbe
ermitteln *73*
setzen *100*
Höhe von Textausgaben *111*
Höhen-/Seitenverhältnis
ändern *99*
Bestimmung des *62*
Bilschirm *72*
Setzen des *53*
Huge (Schlagwort) *16*

I

I/O
siehe Ein-/Ausgabe *115, 126*
imagesize (Bibliotheksfunktion) *86*
initgraph (Bibliotheksfunktion) *86*
Initialisierung
des Grafikpakets *86*
Installation
von Vektor-Fonts *90*
installuserdriver (Bibliotheksfunktion) *89*
installuserfont (Bibliotheksfunktion) *90*
Interrupt-Flag *14*
Interrupts
DOS *126f*
Module und Behandlungsroutinen *35*
IP (Programmzeiger-,
Instructions Pointer) Register *12*

J

Justierung
von Textausgaben *107*

K

keep (Bibliotheksfunktion) *135*
Kommandozeilen-Compiler
Far-Objekte (-zE, -zF und -zH) *25*
Namen für Datensegmente, Optionen *25*
Option -zX *25*
Overlay (-Y) *Option 35*
Overlay (-Yo) Option *34*
Konvertieren der .BGI-Dateien zu
unter OBJ.OBJ-Dateien *51*
Koordinaten
aktuelle Position (CP) *82, 109*
Bezugsecke *48*
maximale *72f*
von Arc ermitteln *72*
Koprozessor
Emulation *40*
Korrekturfaktor für Höhen-/Seitenverhältnis *99*
Kreisbögen *65, 70, 97*
Kreise *67, 71*
Rundungsverhalten bei *53*
Kuchenstück *93, 97*

L

Laufwerke
Funktionen *117*
Lautsprecher
aktivieren *140*

ausschalten *136*
line (Bibliotheksfunktion) *91*
linerel (Bibliotheksfunktion) *91*
linesettingstype (Datentyp) *76, 104*
lineto (Bibliotheksfunktion) *91*
Linien
Art *76, 104*
Dicke *76, 104*
Muster *76*
Parameter *110*
Rechtecke und *95*
Schreibmodus *110*
zeichnen *91*
Linker
Module mit unterschiedlichen Speichermodellen und *28*
Direkte Benutzung des *28*
Löschen
Bildschirm *103*
Bildschirm in der Grafik *67*
Zeichenfenster *67*

M

Makro
Erstellen eines - für Far-Zeiger *27*
MK_FP- *27*
Maßstabfaktor
Grafik *53*
Mathematischer Coprozessor
Überprüfung auf Vorhandensein *122*
MK_FP (Makro in der Laufzeitbibliothek *27*
Modifizierer
Zeiger *24*
Module
Größenbeschränkungen *22*
mit unterschiedlichen Speichermodellen
Linken bei *28*
Monochrom-Adapter *68, 94*
Monochrom-Grafikadapter *65*
moverel (Bibliotheksfunktion) *92*
moveto (Bibliotheksfunktion) *92*

N

near (Schlagwort) *16*
Near-Heap
Größe *9*
nosound (Bibliotheksfunktion) *136*
Null-Flag *14*

O

Objekte
Deklarieren von far- *25*
Klassennamen für far- *25*
Kombinieren von far- in einem Segment *25*
Option pragma und far- *25*
Offsets *16*
Komponenten eines Zeigers *27*
Option pragma
Far-Objekte und die *25*
Out of memory
Fehler *11*
outtext (Bibliotheksfunktion) *92*
outtextxy (Bibliotheksfunktion) *93*
Overflow-Flag *14*
Overlays *137*
Arbeitsweise *30*
Assembler-Routinen und *36*
Aufruf von Routinen über Funktionszeiger *35*
BP-Register und *36*
Debuggen bei *36*
Entwerfen von Programmen für *35*
Expanded Memory und *136*
Extended und Expanded Memory in Verbindung mit *37*
Fehler beim Linken *33*
Größe ändern *10*
Große Programme und *30*
Kommandozeilenoptionen (-Yo) *34*
Linken von *33*
Spechermodelle und *35*
Speicherbelegung bei *31*
Speichermodelle und *32*
Standardgröße der Puffer für *35*
Vorsichtsmaßnahmen *35*

P

palettetype (Datentyp) *80, 98*
Parität *115, 126*
Parität-Flag *14*
Paritätsprüfung *116, 126*
pieslice (Bibliotheksfunktion) *93*
Pixel
Farbe *81, 95*
Polygone

zeichnen *70*
zeichnen und füllen *71*
Ports
siehe Schnittstelle *115, 126*
Programme
Ausführung verzögern *129*
beenden *135*
Overlaytechnik bei sehr großen *30*
residente *135*
Programmsegmentpräfix (PSP)
Größe *10*
Speichermodelle und *11*
Projektdatei
Aufgelistete Bilbliothek in *49*
Prototypen
far- und near-Zeiger und *27*
Module mit unterschiedlichen
Speichermodellen *28*
Puffer
Overlay *10*
Standardgröße der Overlay- *35*
Puffergröße des Grafikpakets *103*
putimage (Bibliotheksfunktion) *94*
putpixel (Bibliotheksfunktion) *95*

R

RAM
Größe *122*
residente Programme *135*
Umgang mit Speicher beim Borland C++
Compiler *11*
ungenutzes *128*
randbrd (Bibliotheksfunktion) *137*
randbwr (Bibliotheksfunktion) *138*
Random-Record-Feld *131f*
Rechtecke *95*
Records
Random *131f*
rectangle (Bibliotheksfunktion) *95*
Register
8086 *12*
Arithmetikoperationen- *12*
AX- *12*
Basisadreß- *13*
BP- *13*
BX- *12*
CS- *16*
CX- *12*
DI- *13*
DS- *16*
DX- *12*
Flag- *12, 13*
für Arithmetikoperationen- *12*
für Index-Operationen *12*
Index- *13*
Overlays und BP- *36*
Programmzeiger IP *12*
Schleifen- und Stringanweisungs- *12*
Segment- *13, 14*
SI- *13*
SP- *13*
Spezial- *13*
registerbgidriver (Bibliotheksfunktion) *95*
registerbgifont (Bibliotheksfunktion) *96*
Registrieren von
Grafiktreibern und Zeichensätze *95, 96*
Schriftarten *56*
Rekursive Funktionen
Speichermodelle und *25*
restorecrtmode (Bibliotheksfunktion) *96*
RS-232 *115, 126*

S

sbrk (Bibliotheksfunktion) *139*
Schnittstelle
serielle *115, 126*
Schreibrichtung
festlegen *107*
Schriftarten
Bitmap- *55*
Clipping von *56*
Datei für *56*
Höhe und Breiten von *56*
Informationen über aktuelle Einstellungen *63*
Vektor- *55*
sector (Bibliotheksfunktion) *97*
Segmente *15*
Adressierungsnotation *15*
Erstellen eines Far-Zeiger in der *27*
Komponenten eines Zeigers *27*
Register *13*
Register *14*
Speicher- *14*
Zeiger *24*
Zeiger und *23*

Seiten
aktive *54*
angezeigte *54*
Puffer *54*
serielle Schnittstelle
Ein-/Ausgabe *115, 126*
setactivepage (Bibliotheksfunktion) *97*
setallpalette (Bibliotheksfunktion) *98*
setaspectratio (Bibliotheksfunktion) *99*
setbkcolor (Bibliotheksfunktion) *100*
setbkcolor (Funktion)
CGA versus EGA *60*
setblock (Bibliotheksfunktion) *139*
setcolor (Bibliotheksfunktion)
größter Farbwert für *77*
setcolor (Bibliotheksfunktion) *101*
setfillpattern (Bibliotheksfunktion) *101*
setfillstyle (Bibliotheksfunktion) *102*
setgraphbufsize (Bibliotheksfunktion) *103*
setgraphmode (Bibliotheksfunktion) *103*
setlinestyle (Bibliotheksfunktion) *104*
setpalette (Bibliotheksfunktion) *105*
setrgbpalette (Bibliotheksfunktion) *106*
settextjustify (Bibliotheksfunktion) *107*
settextstyle (Bibliotheksfunktion) *107*
setusercharsize (Bibliotheksfunktion) *109*
setviewport (Bibliotheksfunktion) *109*
setvisualpage (Bibliotheksfunktion) *110*
setwritemode (Bibliotheksfunktion) *110*
SI-Register *13*
sichtbare Grafikseite *110*
Sign-Flag *14*
sound (Bibliotheksfunktion) *140*
Speicher
Bildausschnitte speichern *76, 86*
DMA *119, 122*
FAR-Heap *131*
FAR-Heap überprüfen *123ff*
freigeben *84, 132*
Grenzen von Paragraphen *15*
Overlays und *31*
Paragraph *15*
Reservierung *52*
Segmente im *15*
Speichermodell und *19*
Umgang mit Speicher beim Borland C++
Compiler *11*
ungenutzer *128*
ungenutzer auf FAR-Heap *129*
Zugriff auf *119*
Speicheradressen
Berechnung von *15*
far-Zeiger und *16*
Near-Zeiger und *16*
Zeiger auf *27*
Speicheradressierung
Segment:Offset-Notation *15*
Standardnotation zur *15*
Speicherbelegung
Datensegment *128*
Speicherblöcke
freie füllen *131*
Größe ändern *139*
random *137, 138*
siehe auch Speicher *132*
Speichermodelle *16f*
Compact- *19*
Definierte *18*
Grafikbibliothek *49*
Größe des near-Heap *9*
Huge- *19*
Large- *19*
Linken bei Modulen mit unterschiedlichen *28*
Medium- *18*
Overlays und *32*
Overlays und *35*
Programmsegementpräfix und *11*
Prototypen in Modulen mit unter schiedlichen *28*
Small- *18*
Speicheraufteilung und *19*
Stack und *11*
Tiny- *18*
Vergleich von *22*
Wechseln von *27*
Windows und *18*
Zeiger und *16, 24*
Speicherplatzprobleme
Out of memory-Fehler *11*
speicherresidente Programme *135*
Spezialregister (8086) *13*
SS-Register *14*
Stack *128*
Größe setzen *9, 11*
Register *13*
Stack-Segment *14*
Standardfarbpalette *73*
Statusabfrage *61*

Statusbits *116, 126*
Stopbits *116, 126*
Strings
  Breite ermitteln *111*
  Clipping *56*
Strukturen
  palettetype *73*
Systemsteuerung, Grafik *50*

T

Tastatur
  Abfrage *116f*
Text
  Informationen über aktuelle Einstellungen *63*
  Justierung *107*
  Parameter für die Ausgabe *107*
  Strings *56*
Textausgabe
  in der Grafik *86f*
  Parameter *81*
  Platzbedarf *111*
  Zeichengröße *109*
textheight (Bibliotheksfunktion) *111*
Textmodus
  Umschaltung von Grafik *96*
textsettingstype (Datentyp) *81*
textwidth (Bibliotheksfunktion) *111*
TLINK (Linker)
  Direkte Benutzung von *28*
Ton
  abschalten *136*
  erzeugen *140*
Trap-Flag *14*
TSR-Programme *135*

U

Überlauf-Flag *14*
Übertragungsparameter *115, 126*
Umgebungsvariable 93 *41*
ungerade Parität *115, 126*
UTIL.DOC *56*

V

Vektor-Fonts
  Größe *109*
Vektor-Zeichensätze
  Installation *90*
  Registrierung *96*
Verzögerung, maschinenunabhängige *129*
Video Graphics Array (VGA)
  Farbsteuerung auf *60*
Video-Adapter
  siehe Grafikadapter *68*
Videomodus
  siehe Bildschirmmodus *96*
viewporttype (Datentyp) *82*
Vorzeichen-Flag *14*
VROOMM *30*

W

Warnton
  ausschalten *136*
  erzeugen *140*
Warnungen
  siehe auch Fehler *140*
Weiteres Carry-Flag *14*
Werte
  Break- *128, 139*
window (Funktion)
  Voreingestelltes Fenster und *48*

X

X-Koordinate *82*
  maximale *78, 79*

Y

Y-Koordinate *82*

Z

Zeichen
  Größe *111*
  in Bildschirmzellen *47*
  Vergrößerungsfaktor *109*
Zeichenfarbe
  ermitteln *73*
  Setzen *101*
Zeichenfenster
  Grenzen *82, 109*
  löschen *67*

setzen *109*
Zeichensätze
Registrierung *96*
Zeichnen
Funktionen für *52*
Zeiger
Arithmetik mit *17*
auf Fehlermeldungen *83*
Deklaration von far- *26*
Deklaration von huge- *26*
Deklaration von near- *26*
Der Vergleichsoperator != bei huge- *17*
Der Vergleichsoperator == bei huge- *17*
Einsetzen von Werten in *17*
far-Speichermodell und *16*
Funktionsprototypen und far- *27*
Funktionsprototypen und near- *27*
huge- *17*
Huge-Speichermodell und *16*
Manipulation von *16*
near-Speichermodell und *16*
Normalisierte *17*
Overhead von huge- *18*
Overlays und *35*
Register und far- *16*
Register und near- *16*
Segment- *23, 24*
Speichermodelle und *16, 24*
Speichermodelle und Far- *26*
Speichermodellgröße und Near- *26*
Stack *13*
Standard-Daten *22*
Vergleich von *17*
Vergleichen von *16*
Wechseln von Speichermodellen und *27*
Zero-Flag *14*
zweidimensionale Balken *66*

Borland

Copyright © 1994, Borland International, Inc. Alle Rechte Vorbehalten. Alle Produktnamen von Borland sind eingetragene Warenzeichen der Borland International Inc.
Deutsche Ausgabe © 1994 Borland GmbH, Monzastraße 4c, D-63225 Langen, Telefon 06103/979-0, Fax 06103/979-290
Zweigniederlassung Schweiz, Solothurnstraße 31, CH-3303 Jegenstorf
Hauptsitz: 100 Borland Way, Scotts Valley, CA 95066-3249, +1 (408) 431-1000
Niederlassungen in: Australien, Belgien, Brasilien, Chile, Deutschland, Dänemark, Frankreich, Großbritannien, Hongkong, Italien, Japan, Kanada, Korea, Lateinamerika, Malaysia, Niederlande, Neuseeland, Schweden, Singapur, Spanien und Taiwan.
• Part # BCP1245GExxxxx